LABORATORY FOR
COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-173

COORDINATION OF PARALLEL PROCESSES IN THE ACTOR MODEL OF COMPUTATION

Nathan Goodman

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Ofran 8½

TS-661

MIT/LCS/TR-173

COORDINATION OF PARALLEL PROCESSES IN THE ACTOR MODEL OF COMPUTATION

Nathan Goodman

December 1976

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

Coordination of Parallel Processes in the Actor Model of Computation

by

Nathan Goodman

Submitted to the Department of Electrical Engineering and Computer Science on May 17, 1976, in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

Two algorithms for the mutual exclusion problem are described and proven to operate correctly. The algorithms are unique in that they use very simple synchronization primitives yet are fair and retain their fairness even if the number of parallel processes in the computer system increases unboundedly over time. One of the algorithms uses simple cells of read/write storage as the primitive; the algorithm is similar to the classic algorithms for this problem proposed by Dijkstra and Knuth, but is generalized to handle an arbitrary number of processes. The second algorithm uses extended cells of storage that model read/modify/write (e.g. test-and-set) instructions. While it is well known how to use read/modify/write instructions to achieve unfair mutual exclusion, their use in a fair algorithm is novel.

The results prove that cells of read/write storage are sufficiently powerful primitives to achieve coordination of parallel processes. There is no theoretical necessity for a model of computation to include more sophisticated synchronization primitives such as semaphores and serializers. But while cells are sufficient, the algorithms are very inefficient; more sophisticated primitives are desirable for that reason.

Thesis Supervisor: Carl E. Hewitt

Title: Associate Professor of Electrical Engineering

TABLE OF CONTENTS

1. Introduction .... Page 2

1.1 Basic Elements .... " 4

2. Busywaiting Synchronization Algorithm Using Cells .... " 18

2.1 The array of cells solution .... " 19

2.2 Formal proof of the solution .... " 38

2.2.1 Handshaking between processes .... " 42

2.2.2 Reformulation of the algorithm using handshakes .... " 56

2.2.3 Proving the theorem using handshakes .... " 72

3. Busywaiting Synchronization Algorithms Using Extended Cells " 96

4. Conclusion .... " 112

Bibliography .... " 115

1. Introduction

In this thesis we present two unique algorithms that solve the mutual exclusion problem and we prove that the algorithms operate correctly. The mutual exclusion problem is typified by a situation in which there is some critical resource that will not work correctly if it is accessed simultaneously by multiple processes. For example the critical resource might be a data-base; if two processes were to modify the data-base at the same time the resulting information could very well be inconsistent. The function of a mutual exclusion algorithm is to coordinate the several processes involved so that no two of them will ever access the resource concurrently. Also, mutual exclusion algorithms are usually required to be fair meaning that all processes that try to access the resource will be allowed to do so eventually. That is, the algorithm must not be able to lock out some or all of the processes indefinitely. Both algorithms that we present are fair.

Mutual exclusion algorithms block the attempts of processes to enter the critical resource while the resource is being referenced by a previous process. The algorithms that we present here are called busywaiting algorithms. This means that when some process that wishes to enter the critical resource is blocked -- i.e. when the process must be prevented from proceeding because the resource is busy -- the process waits in a loop testing the value of some memory location. This is in contrast to devices such as semaphores [Dijkstra, 1968] or serializers [Hewitt, 1975] that block processes by suppending their activation.

The first algorithm that we present uses simple cells of read/write memory as its synchronization primitive. The only instructions that these cells are assumed to implement are update instructions and read-contents instructions. The algorithm is modelled after the classical ones by Dijkstra [Dijkstra, 1965] and Knuth [Knuth, 1966] in that it requires an array of memory cells proportional in size to the number of processes in the system. Unlike the previous work, our algorithm generalizes so as to apply to systems where the number of processes may grow unboundedly over time. We prove the correctness of our algorithm using the actor model of computation.

The second mutual exclusion algorithm that we study uses an extended type of cell as the synchronization primitive; the cells are extended so as to model read/modify/write type instructions that are commonplace in real computers. It is well known how to implement unfair mutual exclusion with read/modify/write instructions: this is the standard test-and-set loop on a binary lock variable. We show how the unfair algorithm can be extended to be fair and how the algorithm may be used in systems with an increasing number of processes.

We shall, in the rest of the Introduction, review the basic elements of the actor model. That section primarily is intended to introduce the syntax and basic definitions we will use in the thesis, and to amplify those elements of the model that are most relevant here.

Chapter 2 presents the cell-based algorithm and a formal proof of its correctness. Chapter 3 studies the extended-cell solution and informally proves that it works properly.

1.1 Basic Elements

The actor model of computation as used in this thesis originated with Carl Hewitt [Hewitt, 1973]; many theoretical issues of the model were extensively developed by Irene Greif in her recent dissertation [Greif, 1975].

Every computational entity in an actor system is an actor. There is no distinction drawn between data and procedures -- both are actors. Information is passed between actors by an operation called message transmission, which is rather analogous to argument passing and returning in conventional systems. It should be noted that "message transmission" does not refer to any sort of inter-process communication; there is only one locus of control in a message transmission and it flows with the message form source to target. As actors are the only entities in an actor system, and since they interact solely by means of message transmissions, therefore message transmissions are the only activities that can take place in an actor system. Message transmissions are called events.

An actor is defined in terms of the messages it accepts and the messages it generates in response. Most actors accept only a narrow class of messages: the addition actor, for example, accepts messages containing a sequence of numbers only; a list actor accepts messages such as 'first', 'rest', and 'cons'; and so forth. If an actor receives a message it doesn't like, it is expected to send an error-message to a special actor called the complaints department. We shan't deal with errors or complaints in any detail in this thesis.

When an actor receives an acceptable message it may generate a very large number of events as a result of this stimulus.* Usually we are not interested in specifying all the messages sent in response; since message transmissions are the only activities that occur amongst actors, a specification of all messages would require specifying the entire actor down to the level of primitives. Instead we just specify a sub-set of the events generated by the actor. The common strategy for suppressing unwanted detail is to ignore most message transmissions, except those whose targets are of interest. Greif calls the set of interesting target actors the "distinguished set". Input/output specifications correspond to a distinguished set containing (1) the actor being called and (2) the actor expected to receive the output. A specification concerned with side-effects might have all cells be in the

* We will see later that in some sense the actor generates the entire future of the process. Here we mean the more limited set of messages obtained by regarding the actor as analogous to a called sub-procedure.

distinguished set. And so forth.

Consider as an example of actor specification, the addition actor. We wish to convey that plus accepts messages containing two numbers and returns their sum as its answer. The specification starts with the message received by plus, and then states what messages result:

Event 1: plus receives a message containing n1 and n2,
where n1 and n2 are both numbers.

Event 2: ? receives a message containing n1 + n2

There is a question-mark in event 2 because we have not stated anywhere the identity of the actor which is to receive the answer. In most programming languages there is an implicit control structure that governs what happens to the results of expressions. Typically, if an expression like (+ 2 3) were embedded in another expression, e.g. (f (+ 2 3)), then the result of (+ 2 3) is implicitly caused to be the argument to f. In the actor model, such control structure is not implicitly present; if one wishes to receive an answer to a message an explicit continuation actor must be present in the message. The activity whereby an actor "returns a value" reduces to just another case of message transmission -- namely sending a message containing the value to the continuation actor.

The plus actor must be defined so as to require that a continuation actor is present in the original message:

If there is an event in the history of the form

event 1: plus receives a message containing n1 and n2
where n1 and n2 are both numbers, and
a continuation, called cont

then there is an event of the form

event 2: cont receives a message containing n1 + n2

This specification is at a high level and says nothing about how plus does its job. The actor may use the hardware add instruction in some cases and multiple precision string addition in other cases, or whatever. There may be many, many events between event 1 and event 2, the specification leaves all that unspecified. But if event 1 does happen then event 2 will happen.

This latter interpretation of the actor definition is most important. The relationship between event 1 and event 2 is called the actor causality relationship: we say, event 1 causes event 2. The "how" of this causality is not specified -- merely that if event 1 happens, then that causes event 2 to happen later.

The generalness of the causal link is reflected in the form of the event statements. Event 2 is the activity of plus answering the caller. Intuitively this event might be described as, "plus sends n1+n2 to cont"; however, the way the event actually described there is no mention of plus at all. This is because plus may have delegated the job of sending the answer to some sub-actor or acquaintance. In a model where control is fully nested -- e.g. in LISP -- the answer would have to be passed back up from the sub-procedure through plus on its way out to the original caller. However, in the actor model the control structure is represented explicitly in message continuations; plus could very easily tell the sub-actor to generate the answer and send it directly to the original continuation.

As it happens, this non-nesting of control occurs commonly in actor definitions. Strictly speaking, a single actor may never generate more than one message in any given process. If the computation at hand requires that several messages be sent -- as in a program of sequential statements or a nested expression -- a whole slew of subsidiary actors are created, one for each message transmission. The final actor created will typically be specified to send the answer to the original caller. No multi-processing is implied by this plethora of actor creation; it is mostly a device for simplifying the formal notation. It also helps avoid problems with the values of local variables. Each actor is born with the values of other actors "frozen in" in a manner similar to the programming language POP-2 [Burstall, Collins, Popplestone, 1971], and so there is no need for such things as stack frames as formal devices, et al.

The only case in which the originally called actor is the one that actually sends the result back happens if the actor does no visible computation. The actor cannot even do any run-time argument type checking. Such an actor must always be a primitive; primitive actors, however, can be of this form and actually do useful things.

Events are the basic computational units in the actor model and we shall refer to them repeatedly. A more convenient syntax for events is desirable therefore. Events will usually be written in the following format:

< target receives message [in activator]* >

*This field is optional, it will be used in multi-process cases.

The format of a message is

(message: sequence-of-arguments
    (reply-to: continuation))

For example, the event of calling plus with the arguments 2 and 3 would be written

event 1: < plus receives (message: [2 3]
                                  (reply-to: cont)) >

and plus's response would be

event 2: < cont receives (message: [5]) >

As we have defined them, actors are devices that map an event into a sequence of events: they map the event whereby they are called into the sequence of message transmissions they cause. An ordered sequence of events is called a behavior. The sequence of events caused by an actor is called the actor's behavior.

If an event corresponds to a procedure call -- i.e. it is of the form

event 1: < procedure receives (message: [--arguments--]
                                          (reply-to: return-pt)) >,

then it will cause many events but eventually, hopefully, there will be an event:

event n: < return-pt receives (message: [--result of procedure--])>

Of course, return-pt is itself an actor and it will map event n into some subsequent events n+1, n+2, ...

For example, consider the following program fragment:

```
            .
            .
            .
      factorial (5) ;
      print ('done') ;
            .
            .
            .
```

The call to procedure factorial corresponds to the event

```
event 1: < factorial receives (message: [ 5 ]
                                  (reply-to: return-pt)) >.
```

Factorial will do many things internally, but assuming it is a well-defined function, it will eventually return to the continuation, return-pt. That event would be

```
event n:   <return-pt receives (apply:[--countdown's value--])>
```

Now, what does the actor return-pt do? factorial has just finished and the next thing the program text says to do is "print ('done');". Furthermore, the program says to do that no matter what factorial did. The actor return-pt must be defined as follows:

```
event 1: < return-pt receives ? >
                causes
event 2: < print receives (message: ['done']
                              (reply-to: return-pt-2))>.
```

The question mark in event 1 means that return-pt cares not what its input is; it does the same thing no matter what. The continuation in event 2 must be an actor like return-pt that performs the step after print ('done') in the program.

Plugging return-pt's specification into the program fragment yields the following scenario:

```
        event 1:  <factorial receives (message: [5]
                                         (reply-to: return-pt))>
                              .
                              .
                              .
        event n:  <return-pt receives (message: [--factorial's value--])>
        event n+1:  <print receives (message: ['done']
                                         (reply-to: return-pt-2))>
                              .
                              .
                              .
```

The dot-dot-dot after event n+1 reflects the fact that program keeps on going. Print will cause many events, eventually return-pt-2 will receive a message and it will cause the next step of the program to be run, etc. To paraphrase an old homily, event n+1 is the first event of the rest of the program's life.

It seems intuitively appealing to break the behavior between events n and n+1. All the events between 1 and n inclusive are reasonably attributable to factorial; they may reasonably be called "Factorial's behavior". The events from n+1 onward are more naturally called "the rest of the program".

This division of behavior is quite useful in many cases. It allows us to talk of an actor's behavior, or the behavior resulting from an event, in a compact and more or less precise way*. We will use this natural terminology often. It is important to note, though, that the division is really arbitrary. The events n and n+1 have no locally observable characteristics that distinguish them from the events before or the events after.

If we decide, therefore, not to break the behavior between events n and n+1, a different interpretation emerges. The behavior resulting from an event may be regarded as all the future behavior of the process. This view interprets behaviors as more than descriptors of the past performance of an actor system; behaviors are also prescriptors of the future of the system.

Almost all actors are pure in the sense that their behavior does not vary of time. Given the same input message at two different times, the actor will cause the same sequence of "next" events both times. If all actors in the system were pure there could never be any time varying behavior in the system -- everytime the system were started up it would inevitably produce the same answer in the same way. There is only one primitive actor

*It is quite hard to make the notion formally precise, though. What is the resulting behavior of an event without a continuation? Even if there is a continuation, we have no assurance that a message will ever be sent to the continuation.

whose behavior is not fixed over time and that is the cell of read/write storage.

Cells are defined to respond to two kinds of messages: one message of the form (message: ['contents?'] (reply-to: continuation)) and the other message (message: ['update' to new-value] (reply-to: continuation)). The first message asks the cell for its contents and the cell responds by sending back the value stored in the most recent update message. The event following a 'contents?' query will always be of the form

$E_{\text{cell's-contents}}$: <continuation receives (message:[cell's-contents])>.

However the actual content of the event will vary as the contents of the cell varies.

Whenever the behavior resulting from an event is time varying that means a side-effect has occurred. By localizing all side-effects to the actions of one particular kind of actor, the cell, reasoning about the time variability of behaviors is greatly simplified. Of course, since actors may be defined by users that utilize arbitrary numbers of cells in arbitrary algorithms, no generality is lost through the simplification.

We have noted that the past behavior of a system together with all the actor definitions in the system prescribe the future course of that system.* Suppose we have an actor system, A, which has been running for a while. A will, therefore, have a behavior B. As long as the actor system is running

* Though if there is parallelism, there may be many possible future courses.

its behavior will continue to grow. If we were to "freeze" the actor system at a point in time, the behavior would of course stop expanding and would have a last event, E_{last}. In order to resume the computation, all we need is E_{last}, because the behavior that results from E_{last} is all the future behavior of the process as described above.

Behaviors are the concrete realization of processes in the actor model, analogous to such things as stack frames in more conventional models of computation. Consider the model suggested by Bobrow and Weigbreit [Bobrow & Weigbreit, 1974] for instance. As a process runs it maintains a push-down stack of stack frames each of which includes a program counter, local variable bindings and all the control structure information necessary for the running procedure to reference non-local variables and to return to its caller. To freeze a process in the Bobrow and Weigbreit model we would stop it after some program step. The stack frame for the running procedure and all its predecessors on the stack are at that point poised, ready to execute the next step in the program.* All that is required to resume the program is to cause the machine to continue executing out of the frozen stack frame.

All models of computation need some method to concretely incarnate running processes. In the Bobrow and Weigbreit model, the stack and most particularly the "current" stack frames play that role. In the actor model, it

* In the model the program counter is assumed to be updated after each program step. For a real running system the stack frames only simulate the model and usually don't update the PC after each instruction.

is behaviors and their last event.

The actor model of computation is largely motivated by an interest in describing systems of multiple processes. The model as developed so far here has dealt only with single process systems, though. The formal machinery developed for the single process case must be extended ever so slightly to embrace multiple processes.

Behaviors are the concrete realization of processes in the actor model. For each process in the actor system there will be a distinct behavior describing its activities since creation. The union of all the individual behaviors is also called a behavior: it is the behavior of the actor system as a whole. If the processes never interact then that is the end of the story. If, however, the processes do interact then we need a little more formal machinery.

If two processes interact we often want to compare an event or events in the behavior of one process with events in the behavior of the other. In order to tell which events go with which process, all events are labelled with a name called an activator. The activator is more or less equivalent to a process name.

Events that include an activator are written

event: <target receives message in activator> .

Our nomenclature for activators will normally be α possibly with a subscript. Phrases like, "event E, in $\alpha_{\text{data-base}}$" will be used as a shorthand for "event E, in the behavior of the process whose activator is α data-base".

Multiple process systems may be implemented in numerous configurations: all the processes may actually run on the same computer with only simulated parallelism; or each process may have its own processor; or some may be one way and some the other way. We wish for our theory, at least in its fundamental form, to be applicable to all forms of parallel processes independent of how the parallelism is achieved. We make no assumptions therefore about the relative speeds of various components of the system. One component might be a human being performing instructions off a written sheet and another component might be an IBM 370/168. The human being might have an effective execution speed of one instruction per second as compared to the machine's 60 million instructions per second.

Also there may be arbitrary and uneven delays between events even in the same process. We may have an algorithm part of which executes on the 370/168 and part of which requires human processing. Though the algorithm represents one single sequence of steps -- i.e. it is a single process -- some events therein are separated by 160 nanoseconds while others are spaced a second apart.

Nor do we assume the existence of a global time standard with which activities may be time-stamped. Global time-stamping of events that transpire in separate processes is feasible for locally executing processes but is often hard to achieve with geographically distributed systems. At a minimum, the existence of a common time standard for all processes requires careful planning ahead, both to acquire the common clocks and to make sure all programs use the time information properly. There are some problems where global time-stamping seems an invaluable aid and others where it adds as many difficulties as it solves.* At this time we are interested in seeing how far asynchronous, non-time based models can go.

The impact of these assumptions is that events in separate processes are not usually comparable -- i.e. it is not usually possible to tell which event happened first. Concretely, the impact of these assumptions is that in general, between any two events in one process there may occur arbitrary numbers of events in other processes.

* Satellite ALOHA networds are a positive example of time-stamping.

2. Busywaiting Synchronization Algorithms Using Cells.

We shall study in this chapter the problem of enforcing mutual exclusion of an arbitrary number of processes with respect to some particular critical or protected actor. The mutual exclusion problem has been investigated exhaustively in the literature. Generally work in this area may be classified according to the primitive synchronization facilities that are assumed to be available. In this context the phrase "primitive facility" means that the operation involved occurs indivisibly, as if it were a single instruction or micro-instruction in the instruction set of a hardware machine.

Naturally the more sophisticated the primitives are that are assumed to exist, the easier it is to solve the mutual exclusion problem and related problems such as the readers/writers problem. Some common synchronization primitives include semaphores [Dijkstra, 1968], monitors [Hoane, 1974], and serializers [Hewitt, 1975]. These primitives all achieve mutual exclusion of arbitrary numbers of processes.

Cells may also be used as the synchronization primitive of a mutual exclusion algorithm as is well known. Dijkstra [Dijkstra, 1965] and Knuth [Knuth, 1966] developed the classic algorithms along these lines. Their algorithms only work, though, if the number of processes in the system does not increase over time beyond a fixed maximum; it remains unproved whether mutual exclusion of an arbitrary number of processes, where the number is not fixed over time, can be achieved using cells. We will prove that it can be accomplished.

We have elected to study mutual exclusion per se because it is the fundamental synchronization activity needed to protect actors from harmful multi-process concurrency. Using a mutual exclusion operator as a building block, other more sophisticated actor protection mechanisms can easily be built. These more sophisticated mechanisms may implement better scheduling algorithms than are possible in the simple mutual exclusion operator; also they may be able to recognize situations where total mutual exclusion of processes is overly restrictive, and they may allow some class of processes to access the protected actor collection concurrently. This latter elaboration corresponds to the well-known readers/writers problem and its extensions.

In the following section we describe the algorithm and demonstrate informally that it works correctly. A formal proof in the actor model is presented in the section following.

2.1 The array of cells solution

The algorithm that we present here is based on the approach proposed by Dijkstra in [Dijkstra, 1965]. The key ingredient of this approach is that the mutual exclusion operator maintains an array of cells that must be at least as large as the number of processes that use the operator. Each element of the array is indexed by the "name" of a process.

The basic form of the Knuth and Dijkstra algorithms is this: When a process wishes to pass through the mutual exclusion guardian it changes the value of its entry in the cell array to indicate to all other processes that it wants to get through. Then some computation is performed on every other entry in the array; the actual computation varies from algorithm to algorithm, but in all cases the purpose of the computation is to indicate whether or not there is another process already executing inside the critical region. Only if this predicate answers that no processes are in the critical region may the current process proceed to pass through the guardian. If the new process is not permitted to enter the critical region now -- i.e. if the computation it performed on the array said, "No!" -- the process must wait. It does so by looping, each time computing the entrance predicate until the answer is "Yes!".

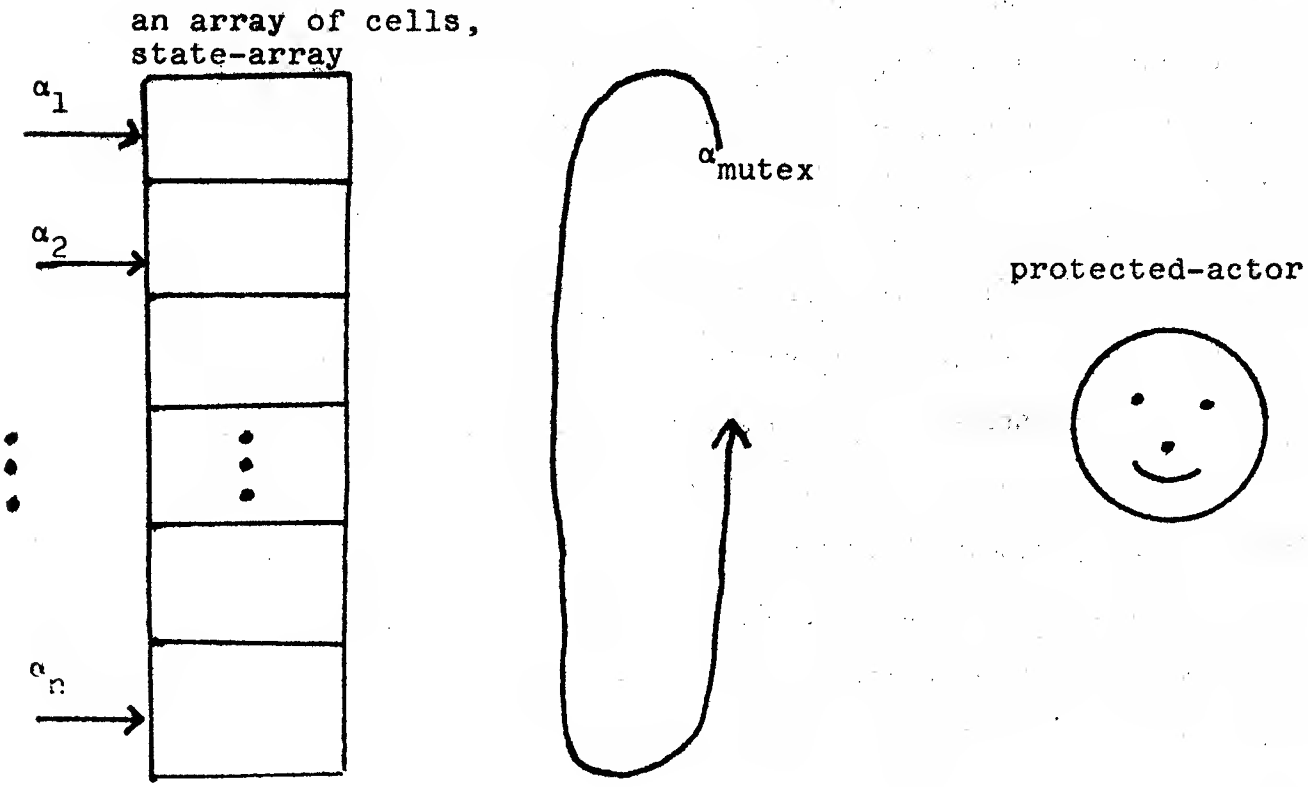
Algorithms that follow this approach have been proven correct by previous researchers. In particular, Greif [Greif, 1975] has proven that a similar algorithm proposed by Knuth [Knuth, 1966] indeed does work. Correctness of mutual exclusion algorithms has two components: First, it must be the case that two processes never execute in the critical region concurrently -- this is a minimum specification; and second, the algorithm must be <u>fair</u> -- i.e. it must be guaranteed that all processes that attempt to pass through the guardian will make it through eventually.

The algorithm of Knuth is known to work, but only if the number of processes in the system does not exceed the size of the array of cells.

We will show how this limitation can be circumvented, thus proving that fair mutual exclusion can be achieved for an arbitrary number of processes using cells as the primitives.

The original algorithms presented by Dijkstra and Knuth are pretty complicated. The algorithm we describe here is similar to theirs in essence, but it is much easier to understand, and much easier to reason about. First we present the simple case of the algorithm where the number of processes is assumed to remain fixed. Then we will extend the approach to handle arbitrary numbers of processes.

Consider the illustrative diagram below:



We will call the array of cells required by the algorithm, <u>state-array</u>; we call the actor being protected, <u>protected-actor</u>. Also, we assume the presence of a special process, α_{mutex}, whose job it is to mind the store. Whenever an external process α_i, wishes to access the protected-actor, it changes the value of state-array [i] and then waits. α_{mutex} in its idle mode loops continually scanning the state-array; eventually α_{mutex} will note the changed value of state-array [i], and inform α_i that it may enter the protected-actor. This algorithm is intended to be fair and we will prove that it is, though by no means is it FIFO. This means that while all processes that try to pass through the mutual exclusion device are assured of getting through, no attempt is made to service requests in the order they are made.

The operation of the device is regulated by the value in <u>state-array</u>. Each element of state-array reflects the state of one process in its efforts to get through the guardian. The elements of state-array range over four values:

idle -- α_i does not wish to enter;
request -- α_i requests permission to enter protected-actor;
grant -- α_i is granted permission to enter protected-actor;
done -- α_i has finished its interaction;

and idle, again, indicating that the interaction is complete from $\alpha_{i's}$ point of view and $\alpha_{mutex's}$ viewpoint, too. It is important that the transition from any state value to the next is always in the province of either α_i or α_{mutex} but never both. The transitions are controlled as follows:

idle to request -- by α_i when it decides to enter;

request to grant -- by α_{mutex} when it allows α_i to enter;

grant to done -- by α_i when it has finished with protected-actor;

done to idle -- by α_{mutex} when it notices that α_i is done.

Each external process must follow an established protocol in its dealings with the protected-actor. The mutual exclusion operator can only be assured of working properly if the protocol is adhered to dutifully every time the protected-actor is referenced. In order to localize the implementation of this protocol, we will utilize the concept of "encasement" put forward by Greif [Greif, 1975] and Hewitt [Hewitt, 1974].

We will imagine that the actor being protected is fully enclosed, encased, within another actor that serves as an alias for it. All other actors know only the alias; they do not know the protected-actor itself. Whenever an actor wishes to send a message to the protected-actor, it sends it instead to the alias-protected-actor; the alias observes the protocol, and retransmits the message to its ward when the protocol allows it. Alias-protected-actor is specified by the algorithm written descriptively below (the algorithm is specified formally in LISP shortly hereafter):

```
alias-protected-actor ≡
(1) receive argument, and call it the-input-message
(2) set local identifier i = the name of the process
(3) update state-array [i]: = 'request'
(4) loop waiting for state-array [i] to be 'grant', as follows
```

```
(5) ask state-array [i] for its contents, and let state = the contents
(6) there are two cases for state:
    (6-1) state = 'request' -- repeat from step (4)
    (6-2) state = 'grant' -- proceed with step (7).
(7) send message (message: the-input-message (reply-to: step -(8)-below))
                 to protected-actor
(8) when finished referencing protected-actor, update state-array [i]:='done'
(9) loop waiting for state-array [i] to be 'idle', as follows
(10) ask state-array [i] for its contents, and let state = the contents
(11) there are two cases for state:
    (11-1) state = 'done' -- repeat from step (9)
    (11-2) state = 'idle' -- exit to externally supplied continuation
               with answer received from protected-actor in step (8).
END:
```

The storekeeper process, α_{mutex}, follows a different regimen. α_{mutex} may be thought of as running in two modes, a scan mode and a wait mode. In the scan mode, α_{mutex} circulates through the state-array scanning each element in turn. When it reads a state-array[i] whose value is 'request', it stops its scan. It changes the element, state-array[i] to be 'grant', and enters its wait mode. In wait mode, α_{mutex} loops testing the same element of the array over and over. When that element becomes 'done', α_{mutex} changes it back to 'idle', and resumes scanning. Importantly, α_{mutex} always resumes its scan with the array element after state-array[i].

The algorithm that α_{mutex} executes is embodied in the actor described below, an actor that we call regulate-mutual-exclusion. Steps (2), (3), (9), (10), and (11) comprise the scan mode of the algorithm; steps (14) - (18) implement the wait mode. (This actor is specified formally in LISP following the description).

```
regulate-mutual-exclusion ≡
(1) set cell i:=  first process name known
(2) ask state-array[i] for its contents, and set state = contents
(3) there are two cases for state:
    (3-1) state = 'idle' -- go to end of loop, step (9), to continue scan
    (3-2) state = 'request' -- proceed with step (4)
(4) update state-array[i]:=  'grant'
(5) loop waiting for state-array[i] to be 'done', as follows
(6) ask state-array[i] for its contents, and set state = contents
(7) there are two cases for state:
    (7-1) state = 'grant' -- repeat from step (5)
    (7-2) state = 'done' -- proceed with step (8)
(8) update state-array[i]:=  'idle'
(9) resume or continue scanning the state-array as follows
(10) there are two cases for the process name index, i:
    (10-1) i = last process name known -- update i:=  first process
                                                       name known
    (10-2) else, update i:= next process name after i
(11) repeat from step (2).
END
```

The two actors alias-protected-actor and regulate-mutual-exclusion specified formally in a LISP-like notation on the following pages. A certain license with LISP syntax is taken in that we write array references in the ALGOL-ish form

```
        array [index]
```

rather than the LISP

```
        (get 'array index)
    and (store 'array index value).
```

```
(defun alias-protected-actor (the-input-message)
      (prog (i state answer)

      Step-2    (setq i (the-name-of-the-process))
      Step-3    (set state-array[i] 'request')
      Step-4    ;; loop waiting for state-array[i] to be 'grant'
      Step-5    (setq state state-array[i]
      Step-6    (cond
      Step-6-1      ((equal state 'request') (goto step-4))
      Step-6-2      ((equal state 'grant') (goto step-7))
                    (else (error))
      Step-7    (setq answer (protected-actor the-input-message))
      Step-8    (set state-array[i] 'done')
      Step-9    ;; loop waiting for state-array[i] to be 'idle'
      Step-10   (setq state state-array[i])
      Step-11   (cond
      Step-11-1     ((equal state 'done') (goto step-9))
      Step-11-2     ((equal state 'idle') (return answer))
                    (else (error)))))
```

```
(defun regulate-mutual-exclusion nil
      (prog (i state)
      Step-1  (setq i (first-process-name-known))
      Step-2  (setq state state-array[i])
      Step-3  (cond
      Step-3-1     ((equal state 'idle')(goto step-4))
                    (else (error)))
      Step-4  (set state-array[i] 'grant')
      Step-5  ;; loop waiting for state-array[i] to be 'done'
      Step-6  (setq state state-array[i])
      Step-7  (cond
      Step-7-1     ((equal state 'grant')(goto step-5))
      Step-7-2     ((equal state 'done')(goto step-8))
                    (else (error)))
      Step-8  (set state-array[i] 'idle')
      Step-9  ;; resume or continue scanning the state-array
      Step-10 (cond
      Step-10-1    ((equal i (last-process-name-known))
                             (setq i (first-process-name-known)))
      Step-10-2     (else (setq i (next-process-name-after i))))
      Step-11 (goto step-2)))
```

The proper operation of this mechanism is apparent assuming that the system starts out in its "natural" initial condition. That is, all entries of the state-array must be initialized to 'idle', no external process may be referencing protected-actor initially, and α_{mutex} must start executing regulate-mutual-exclusion at step (1). We shall explain the correctness of this solution informally at this time; a formal proof is presented in the next section.

There are two aspects to the correct operation of a fair mutual exclusion operator. First, does it even implement mutual exclusion -- i.e. does it prevent the simultaneous access of two processes to the protected-actor. The second aspect is the fairness of the device; will every process that attempts to reference the protected-actor be allowed to do so eventually.

We will demonstrate first that the system here does indeed achieve mutual exclusion. Proceeding from the stated initial conditions, it is clear that no process will ever reference the protected-actor unless alias-protected first sees the state-array element equal to 'grant'. Also, after the process is done, and not before, alias-protected-actor changes the 'grant' state to 'done'. Thus if no other actor ever modifies a state-array element that equals 'grant', we may be sure that no process will access the protected-actor unless its state-array element is 'grant'.

Furthermore, each process that wishes to enter the protected region first sets its state element to be 'request' and does not set it to be 'grant'. The only actor that does set states to 'grant' is the actor regulate-mutual-exclusion, that actor will only set one state to 'grant' and no more until that state-array element cycles to 'done' first. It follows that no more than one state-array element will equal 'grant' concurrently and, therefore, that no more than one process will reference protected-actor concurrently.

Fairness of the operator may be inferred from the scan algorithm employed, assuming that the operations in step (10) of regulate-mutual-exclusion are all well defined, one-to-one functions. That is, there must be a unique first process name, a unique last process name, and the function next applied iteratively starting with the first name must yield all the names known to the system exactly once. Given all this, it is clear that if no process ever requests entry then each state-array element will be scanned once before any element is scanned twice.

If one process sets its state-array element to 'request', regulate-mutual-exclusion will note the fact the next time that element is scanned. The scanning will then be interrupted and steps (4) - (8) attempted, and the state-array element will be set to 'grant'. The specification of alias-protected-actor makes it clear that once state-array[i] is set to 'request', the entire sequence of the protocol must inevitably occur. Thus it is a foregone conclusion that the state element will eventually become 'done'. Regulate-mutual-exclusion detects the 'done' state and resets it to 'idle' thus completing the cycle.

The important fact is that the entire interlude from step (4) to step (8) does not affect the scan parameter, i. Thus providing that all the steps (4) - (8) do occur the scan will resume as if there had been no interruption. And as we have noted, the interaction between the two processes does insure the completion of those steps. Thus α_{mutex} will scan all the other elements of state-array before it scans the one just let through again. Therefore all the elements of the array will get a "next" shot through to the protected-actor and none can be locked out. I.e. the operator is fair.

<u>This algorithm for mutual exclusion only works so long as the number of processes that may wish to access the protected-actor does not exceed the size of the state-array</u>. Of course, nothing in either alias-protected-actor nor regulate-mutual-exclusion prohibits the use of a variable size structure in place of an actual fixed size array. Suppose state-array were physically implemented as a list. Whenever a new process sought to join the crowd of processes with rights to the mutual exclusion operator, a new entry could be cons'ed onto the front of the state-array list, and the variable pointing to the front of the list could be updated to include the new entry.

In this model the identifier "state-array" will be used to name the list of state-array elements. The notation -- state-array[i] -- must be understood as a symbolically indexed reference into the list pointed at by state-array. We may continue to assume that the expression state-array[i] returns a pointer to the cell that the i<u>th</u> process must twiddle in order to pass through the mutual-exclusion operator; therefore the statements like

```
(set state-array[i] 'request')
```

that appear in the actor alias-protected-actor will still work even though state-array is changed to a list.

The algorithm for regulate-mutual-exclusion though references all the entries of state-array sequentially and it is more convenient to rewrite that actor using car's and cdr's than it is to try to make the array notation work. A version of regulate-mutual-exclusion that is particularized for the case of state-array being a list is presented on the following page:

```
(defun regulate-mutual-exclusion nil
      (prog (i-state state)
            (setq i-state state-array)
    Step-2  (setq state (car i-state))
            (cond
                  ((equal state 'idle')(goto step-9))
                  ((equal state 'request')(goto step-4))
                   (else (error))
    Step-4  (rplaca i-state 'grant')
            ;; loop waiting for state-array[i], i.e. i-state, to be 'done'
    Step-6  (setq state (car i-state))
            (cond
                  ((equal state 'grant')(goto step-6))
                  ((equal state 'done')(goto step-8))
                   (else (error))
    Step-8  (rplaca i-state 'idle')
            ;; resume or continue scanning the state-array
    Step-9  (cond                    ; if at end of state-array list
                   ((null (cdr i-state))  ; then reset i-state to beginning
                          (setq i-state state-array)
                    (else (setq i-state (cdr i-state))))
                                          ; else set i-state to next entry
             (goto step-2)))
```

The operation of creating a new process may add the new process to the state-array by cons'ing a cell for the new process onto state-array. This procedure is described in algorithms for actors fork and expand-state-array below:

```
fork ≡
(1)  receive argument and call it new-process
(2)  do whatever has to be done in the innards of the system to create
     a new process
(3)  expand-state-array for the new-process (see below)
(4)  exit to externally supplied continuation
END
```

```
expand-state-array ≡
(1)  receive argument and call it new-process
(2)  allocate a new cell and call it new-state.  Update new-state's
     initial contents to be 'idle'
(3)  Cons new-state onto state-array and let new-state-array = the
     returned value
(4)  store new-state in the bowels of the system in a manner associated
     with new-process
(5)  update state-array:= new-state-array
(6)  exit
END
```

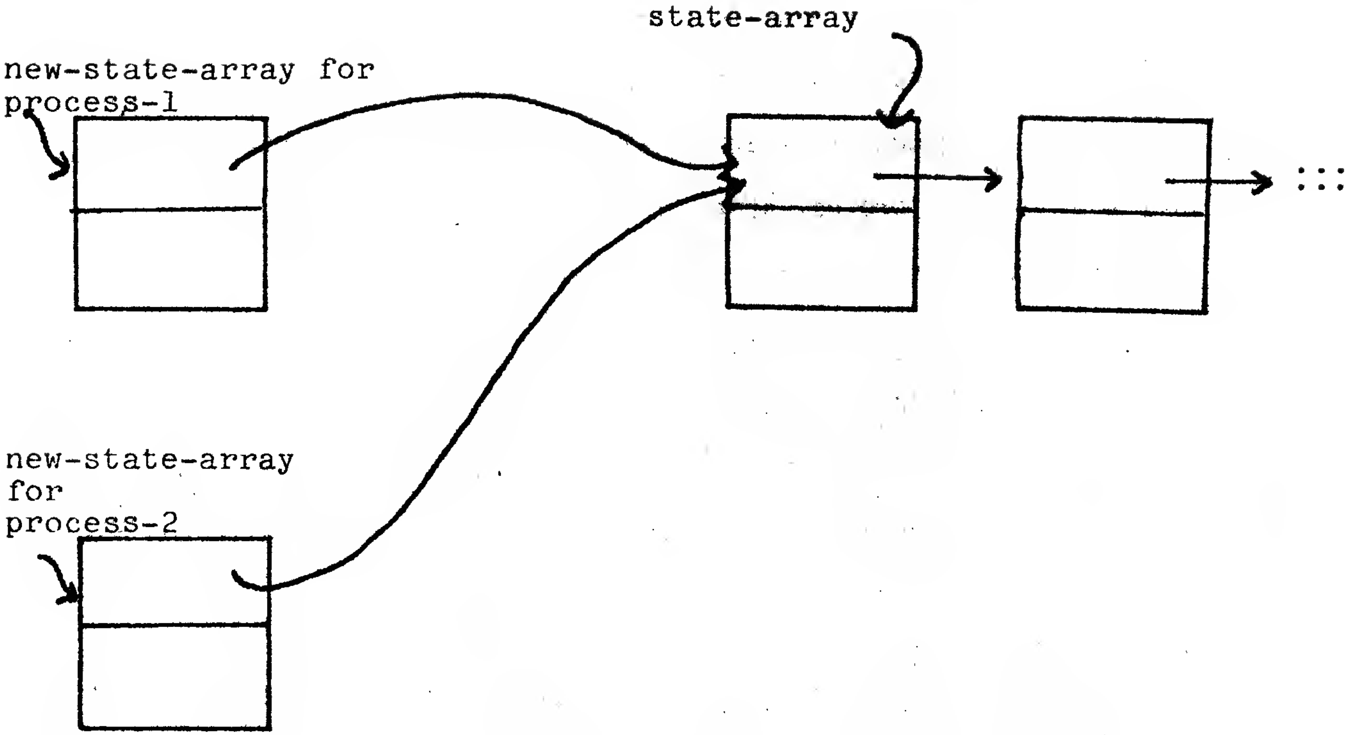
The actor expand-state-array is expressed formally in LISP on the following page:

```
(defun expand-state-array (new-process)
          (prog (new-state new-state-array)
     Step-2     (setq new-state 'idle')
     Step-3     (setq new-state-array (cons new-state state-array))
     Step-4     (... store new-state in the system...)
     Step-5     (setq state-array new-state-array)))
```

There is a potential timing error associated with the actor expand-state-array if it is executed concurrently by multiple processes. A possible behavior involving two concurrent executions of the actor is illustrated below:

```
          Process-1                              Provess-2
     (2)  new-state := 'idle'                (2) new-state := 'idle'
     (3)  new-state-array :=                 (3) new-state-array :=
              (cons new-state state-array)           (cons new-state state-array)
     (4)  ...                                (4) ...
     (5)  state-array := new-state-array     (5) state-array := new-state-array
```

That is, both processes could create the new-state-array using the same previous state-array resulting in a structure like



Whichever process updates state-array in step (5) last is the one that will win out in the end. Its new-state cell will be included in the state-array list; the other entry, while not garbage nor possessing a dangling reference, will never be referenceable from state-array.

This bug would be avoided, however, if the operation of adding processes to state-array were a mutually exclusive operation. There would be no problem with extending the state-array list to arbitrary size, if only the operation were restricted to one process at a time. In the actor model, new processes do not arise through spontaneous generation; aside from the processes that are stipulated as existing in the system's initial conditions, all other processes are created by events. Events, of course, are activities in some process that already exists.

If we assume that all initial processes are represented in the state-array then we may specify that process creation occurs in a mutually exclusive manner by protecting the actor fork with precisely the mutual exclusion operator that we have described here. That is we may define an actor alias-fork that is identical to alias-protected-actor, except it relays the input messages to fork instead of protected-actor. In other words we may use state-array to protect the actor that expands state-array!

There is one other trouble-spot in extending the "array of cells" solution to arbitrary number of processes -- the processes being added to the state-array may be very prolific and may themselves create more new processes. These additional new processes will have to be added to the state-array, too, of course. And they too may be very prolific.

Suppose that when new processes are added to state-array that they are added at the end. And suppose that when each passes through to the protected region it creates a new process that immediately attempts to pass through the mutual exclusion operator itself. Under these conditions, the scan may be stuck in an infinitely growing morass of fast breeding processes. Any requests entered closer to the beginning of the array would never be served.

This bug may be avoided, though, by the simple strategen of extending the array at its front. Now, even though the array might grow without bound over time, every request that is entered would be scanned and allowed through eventually. This is because no process created during one scan of the state-array would itself be attended until the next scan.

Thus, the "array of cells" solution to the mutual exclusion problem may be extended to handle the most general case of unbounded numbers of processes.

2.2 Formal proof of the solution

We shall prove the correctness of the algorithm in two stages. First, we will prove that mutual exclusion per se is implemented; then we will prove the fairness of the algorithm. For the first part it doesn't matter whether or not the number of processes remains constant. The extension to handle arbitrary numbers of processes need only be considered in the fairness proof.

Before proceeding, let us state precisely what is being proved.

Definition: a normal returning actor

Let protected-actor be an actor which includes in its specifications the following fact: if the event

$E_{enter-i}$: <protected-actor receives
(message: any-message
(reply-to: continuation)) in α_i>

appears in the behavior then the event

E_{exit-i}: <continuation receives
(message: any-answer) in α_i>

will appear later in the behavior.

Then protected-actor will be called a normal returning actor.

This first definition ensures that the actors being protected are all well-behaved and act like normal sub-routines. If we send a message to the protected-actor we expect it to answer and not fly off on its own someplace.

Definition: mutually exclusive reference

Let protected-actor be a normal returning actor.

Then protected-actor is said to be referenced in a mutually exclusive fashion if and only if for all quadruples of events

$$(E_{enter-i}, E_{exit-i}, E_{enter-j}, E_{exit-j}), j=i$$

one of the following two orderings holds:

either (1) $E_{enter-i}$ before E_{exit-i} before $E_{enter-j}$ before E_{exit-j}

or (2) $E_{enter-j}$ before E_{exit-j} before $E_{enter-i}$ before E_{exit-i}

Definition: fair encasement

Let protected-actor be a normal returning actor. And let alias-protected-actor be another actor.

Alias-protected-actor fairly encases protected-actor if and only if a behavior has the following event in it:

$E_{hello-i}$: <alias-protected-actor receives
(message: any-message
(reply-to: continuation)) in α_i>

Then it also has the following events in the stated order:

$E_{enter-i}$: <protected-actor receives
(message: any-message
(reply-to: alias-continuation)) in α_i>

E_{exit-i}: <alias-continuation receives
(message: any-answer) in α_i>

$E_{byebye-i}$: <continuation receives
(message: any-answer) in α_i>.

Definition: fair mutual exclusion actor

Let protected-actor and alias-protected-actor be actors and suppose that alias-protected-actor fairly encases protected-actor.

Then alias-protected actor is a fair mutual exclusion actor for protected-actor if and only if for all histories which contain

$E_{hello-i}$, $E_{enter-i}$, E_{exit-i}, $E_{byebye-i}$, $E_{hello-i}$, $E_{enter-j}$, E_{exit-j}, and $E_{byebye-j}$ $i \neq j$

one of the following two orderings holds:

either (1) $E_{enter-i}$ before E_{exit-i} before $E_{enter-j}$ before E_{exit-j}

or (2) $E_{enter-j}$ before E_{exit-j} before $E_{enter-i}$ before E_{exit-j}.

In particular these orderings hold even if $E_{hello-j}$ were between $E_{hello-i}$ and $E_{byebye-i}$ or the converse.

The theorem we shall prove is this:

Theorem: Given the actor alias-protected actor as specified in section 2.1 above and any actor protected-actor which satisfies the constaints in the definition here. Also, given the actor regulate-mutual-exclusion and the process α_{mutex} as specified in section 2.1

Suppose that the specified system starts out in the following initial conditions:

(1) each element of state-array equals 'idle';

(2) no event of the form $E_{enter-i}$ has yet occurred;

(3) α_{mutex} has not yet begun to execute regulate-mutual-exclusion, but it will execute the actor from the beginning once the system is started up.

Then, alias-protected-actor is a fair mutual exclusion actor for protected-actor.

The proof of this theorem will be facilitated by the concept of an "inter-process handshake" describing the interaction between the external process and α_{mutex}. The actors alias-protected-actor and regulate-mutual-exclusion interact by means of a protocol with the following character: one process sets the cell, state-array[i], to some particular state value and then busywaits until state-array[i] changes to some other value. The other process meanwhile is already set to look for a particular value in state-array[i]; when it sees that value it "shakes hands" with the first process by causing the next transition of state-array[i]. In this way the two processes coordinate each other's activities and lead each other through the algorithm in a step-by-step sequential manner.

In the following sub-section we will formalize the concept of inter-process handshaking. After that the mutual exclusion algorithm will be reformulated in terms of handshakes and we will use the reformulated version to prove the main theorem.

2.2.1 Handshaking between processes

The concept of handshaking will be defined by specifying an actor that implements it. Then we prove several useful theorems regarding the properties of handshake actors.

The actor handshake receives messages of the form

```
(message: (shake: cell
          (set-to: value-1)
          (then-wait: value 2))
          (reply-to: continuation)).
```

handshake will update the cell to value-1 then loop busywaiting until the cell's contents become value-2. At that time, handshake replies to the continuation. The algorithm for handshake is described below. A formal version of the actor in LISP follows it.

```
Handshake ≡
(1) receive input message of the form
```

```
(message: (shake: cell
           (set-to: value-1)
           (then-wait: value-2))
           (reply-to: continuation))
```

```
(2) update the contents of the cell to value-1.
(3) busywait for the contents of the cell to become value-2 as follows:
(4) read contents of cell and let state = the contents.
(5) there are two cases for state:
       (5-1) state = value-2 -- repeat from step (3);
       (5-2) state = value-2 -- proceed with step (6).
(6) exit to the continuation
END.
```

The actor is specified formally below:

```
(defun handshake (shake-cell value-1 value-2)
      (prog (state)
Step-2     (setq shake-cell value-1)
Step-4     (setq state shake-cell)
Step-5     (cond ((not (equal state value-2))(goto step-4)))))
```

Definition: completion of a handshake

A handshake is completed when a reply to the continuation occurs. That is, given the event

$E_{\text{handshake-1-2}}$: <handshake receives
(message: (shake: cell
(set-to: value-1)
(then-wait: value-2))
(reply-to: continuation-1))in α_a>.

$E_{\text{handshake-1-2}}$ is completed by the next event, if any, of the form

$E_{\text{complete-1-2}}$: <continuation-1 receives
(message: ?) in α_a> .

<u>Definition</u>: <u>matching a handshake</u>

Let $E_{\text{handshake-1-2}}$ be an event of the form

<handshake receives
(message: (shake: cell
(set-to: value-1)
(then-wait: value-2))
(reply-to: continuation-1)) in α_a>.

Let $E_{\text{update-2}}$ be an event of the form

<cell receives
(message: ['update' to value-2]
(reply-to ?)) in α_b>.

Then $E_{\text{update-2}}$ is said to match the handshake $E_{\text{handshake-1-2}}$.

The reader should note that matching of handshakes is purely a syntactic matter. The matching event is not in any sense guaranteed to satisfy the busywait loop in the handshake and thus lead to the completion of the handshake; indeed an event that matches a handshake may even occur before the handshake!

Definition: completion-causing event of a handshake

Given $E_{handshake-1-2}$ as in the previous definitions, and given an event $E_{update-2}$ that matches $E_{handshake-1-2}$.

Consider the event

```
<cell receives
       (message: ['update' to value-1]
                  (reply-to: step (3) of handshake)) in α_a>
```

occurring after $E_{handshake-1-2}$ but before the reply to continuation-1; call this event $E_{set-to-1}$.

Consider the class of events

```
<cell receives
        (message: ['contents?']
                   (reply-to: step (4) of handshake)) in α_b>
```

also occurring after $E_{handshake-1-2}$ but before its completion. Call these events $E_{wait-for-2}$.

Let E_{clobber} be any event of the form

E_{clobber}: <cell receives
(message: ['update' to not-value-1]
(reply-to: ?)) in $\alpha_?$>

Where not-value-1 = value-1.

The event $E_{\text{update-2}}$ is a completion-causing event of $E_{\text{handshake-1-2}}$ if and only if $E_{\text{update-2}}$ appears in the behavior after $E_{\text{handshake-1-2}}$, and no event of the form E_{clobber} is between $E_{\text{update-2}}$ and the next occurrence of $E_{\text{wait-for-2}}$.

Theorem: Completion-causing event causes completion of a handshake [This theorem expresses the fact that completion-causing events are aptly named; that a handshake will complete if and only if a completion-causing event is present.]

Given an event $E_{\text{handshake-1-2}}$ as above and any event $E_{\text{update-2}}$ that is a completion-causing event of $E_{\text{handshake-1-2}}$.

Then $E_{\text{handshake-1-2}}$ will be completed -- i.e. handshake will reply to the continuation -- if and only if some event $E_{\text{update-2}}$ is present in the behavior.

Proof: This result follows directly from the specified algorithm for handshake and the axioms for cells. Handshake will reply to the continuation in step (6) of the algorithm if and only if it read the contents of the cell via an event of the form $E_{wait-for-2}$ and found the contents to be value-2.

From the axioms of the cell we see that this condition requires that the most recent update event in the target ordering of the cell must be of the form $E_{update-2}$.

A simple but important corollary of this theorem applies when the "set-to" value does not equal the "then-wait" value of a handshake. First we define a bit of terminology.

Definition: proper handshakes

Let $E_{handshake-1-2}$ be a handshake event as above of the form

```
E_handshake-1-2: <handshake receives
                          (message:(shake: cell
                                   (set-to: value-1)
                                   (then-wait: value-2))
                          (reply-to: continuation)) in α >
```

$E_{handshake-1-2}$ is called a proper handshake if and only if value-1 ≠ value-2.

Corollary: completion causing event of a proper handshake

Let $E_{handshake}$ be a proper handshake event in α_h.

Then no completion causing event of $E_{handshake}$ can be an event in α_h, also.

Proof: All completion causing events of $E_{handshake}$ must be events that update its cell between $E_{handshake}$ and the completion of the handshake; also the event must update the cell to value-2.

The only event in the handshake that updates the cell is the "set-to:" event therein and that event updates the cell to value-1. Since value-1 $\neq$ value-2, the "set-to:" event cannot be a completion causing event of the handshake. And since no events occur in α_h other than events that pertain to the handshake until its completion, no events in α_h can cause the completion.

The property expressed by the corollary is important because it ensures that proper handshakes do cause the running process to wait for some specified events in another process. Hereafter we shall assume that all handshakes are proper.

Another interesting property of handshakes is that they may be chained one after the other resulting in a multi-process sequencing of events as we will see shortly. First we need to make definitions similar to the ones above but involving inter-process handshaking instead of simple updating of events.

Definition: a matching handshake

Let $E_{handshake-1-2}$ be an event of the form

```
E_handshake-1-2: <handshake receives
                          (message: (shake: cell
                                    (set-to: value-1)
                                    (then-wait: value-2))
                                    (reply-to: continuation-1)) in α_a>
```

Let $E_{\text{handshake-2-3}}$ be an event of the form

```
<handshake receives
              (message: (shake: cell
                         (set-to: value-2)
                         (then-to: value-3))
                         (reply-to: continuation-2)) in α_b>.
```

Then, $E_{\text{handshake-2-3}}$ is said to be a matching handshake for $E_{\text{handshake-1-2}}$.

<u>Definition</u>: <u>completion-causing handshake</u>

Given $E_{\text{handshake-1-2}}$ and $E_{\text{handshake-2-3}}$ as in the previous definition. Let $E_{\text{set-to-3}}$ and $E_{\text{wait-for-2}}$ be events related to $E_{\text{handshake-1-2}}$ as in the definition of a completion causing event of a handshake. And let E_{clobber} be any event that updates the cell to a value not equal to value-2.

Then, $E_{\text{handshake-2-3}}$ is a completion-causing handshake of $E_{\text{handshake-1-2}}$ if and only if the following conditions hold:

(1) $E_{\text{handshake-2-3}}$ is after $E_{\text{set-to-1}}$ and before the completion of $E_{\text{handshake-1-2}}$;

and (2) There are no events E_{clobber} between $E_{\text{set-to-1}}$ and the completion of $E_{\text{handshake-1-2}}$.

<u>Theorem</u>: completion-causing handshakes cause completion

Given an event $E_{\text{handshake-1-2}}$ as above and let $E_{\text{handshake-2-3}}$ be any completion causing handshake of $E_{handshake-1-2}$.

Suppose also that there is no other event $E_{update-2}$ of the form

$E_{update-2}$: <cell receives
(message: ['update' to value-2]
(reply-to: ?)) in $\alpha_?$>

between the event $E_{set-to-1}$ (related to $E_{handshake-1-2}$ as in the previous definitions) and the completion of $E_{handshake-2-3}$.

Then, $E_{handshake-1-2}$ will be completed -- i.e. handshake will reply to the continuation -- if and only if some event $E_{handshake-2-3}$ appears in the behavior.

Proof: Let $E_{set-to-2}$ represent the class of events between each $E_{handshake-2-3}$ iand its completion wherein the cell is updated to value-2. I.e. each $E_{set-to-2}$ is of the form

$E_{set-to-2}$: <cell receives
(message: ['update' to value-2]
(reply-to: step (3) of handshake))in $\alpha_?$>.

By supposition the events $E_{set-to-2}$ are the only events between $E_{set-to-1}$ and the completion of $E_{handshake-1-2}$ that update cell to any value whatsoever. Not all the events $E_{set-to-2}$ need occur in the range between $E_{set-to-1}$ and the completion but any that do are completion causing events of $E_{handshake-1-2}$, and thus fulfill the "if" part of the theorem.

The "only if" part follows from the observation that those $E_{\text{set-to-2}}$ events that are completion causing events of $E_{\text{handshake-1-2}}$ are its only completion causing events.

A useful corollary to the above theorem applies to chains of completion causing handshakes. Given a sequence of handshake events where each handshake but the first causes the completion of its predecessor; if the first handshake does in fact occur, then all of the handshakes except the last one will complete one by one in sequence. Though the handshakes appear in separate processes it is as if there were an activator type causal link between them.

Definition: chains of completion-causing handshakes

Let $E_{\text{handshake-1-2}}$, $E_{\text{handshake-2-3}}$, $E_{\text{handshake-3-4}}$, ..., $E_{\text{handshake-n-m}}$ each be handshake events. Let $E_{\text{handshake-1-2}}$, $E_{\text{handshake-3-4}}$, ... be events in process α_a and let the others be in process α_b. $\alpha_a \neq \alpha_b$. Suppose further that $E_{\text{handshake-1-2}}$ is before $E_{\text{handshake-3-4}}$ is before $E_{\text{handshake-5-6}}$ ... in the α_a activator ordering and that the events of α_b are ordered similarly.

Let $E_{\text{handshake-2-3}}$ be a completion-causing handshake of $E_{\text{handshake-1-2}}$; let $E_{\text{handshake-3-4}}$ be a completion causing handshake of $E_{\text{handshake-2-3}}$; etc.

Finally assume that there are no events updating the handshake cell other the "set-to:" events in the handshakes, between $E_{\text{handshake-1-2}}$ and the event $E_{\text{set-to-n}}$, where $E_{\text{set-to-n}}$ is the "set-to:" event in $E_{\text{handshake-n-m}}$.

Then, the sequence of events $E_{\text{handshake-1-2}}$, $E_{\text{handshake-2-3}}$, $E_{\text{handshake-3-4}}$, ..., $E_{\text{handshake-n-m}}$ is called a chain of completion-causing handshakes.

Corollary: chain of completion-causing handshakes

Given a chain of completion-causing handshakes $E_{handshake-1-2}$, $E_{handshake-2-3}$, ..., $E_{handshake-n-m}$.

Then each of the handshakes but the last one will complete in sequence -- i.e. $E_{handshake-1-2}$ will complete before $E_{handshake-2-3}$ which will complete before $E_{handshake-3-4}$, etc. $E_{handshake-n-m}$ will not complete, but it will start.

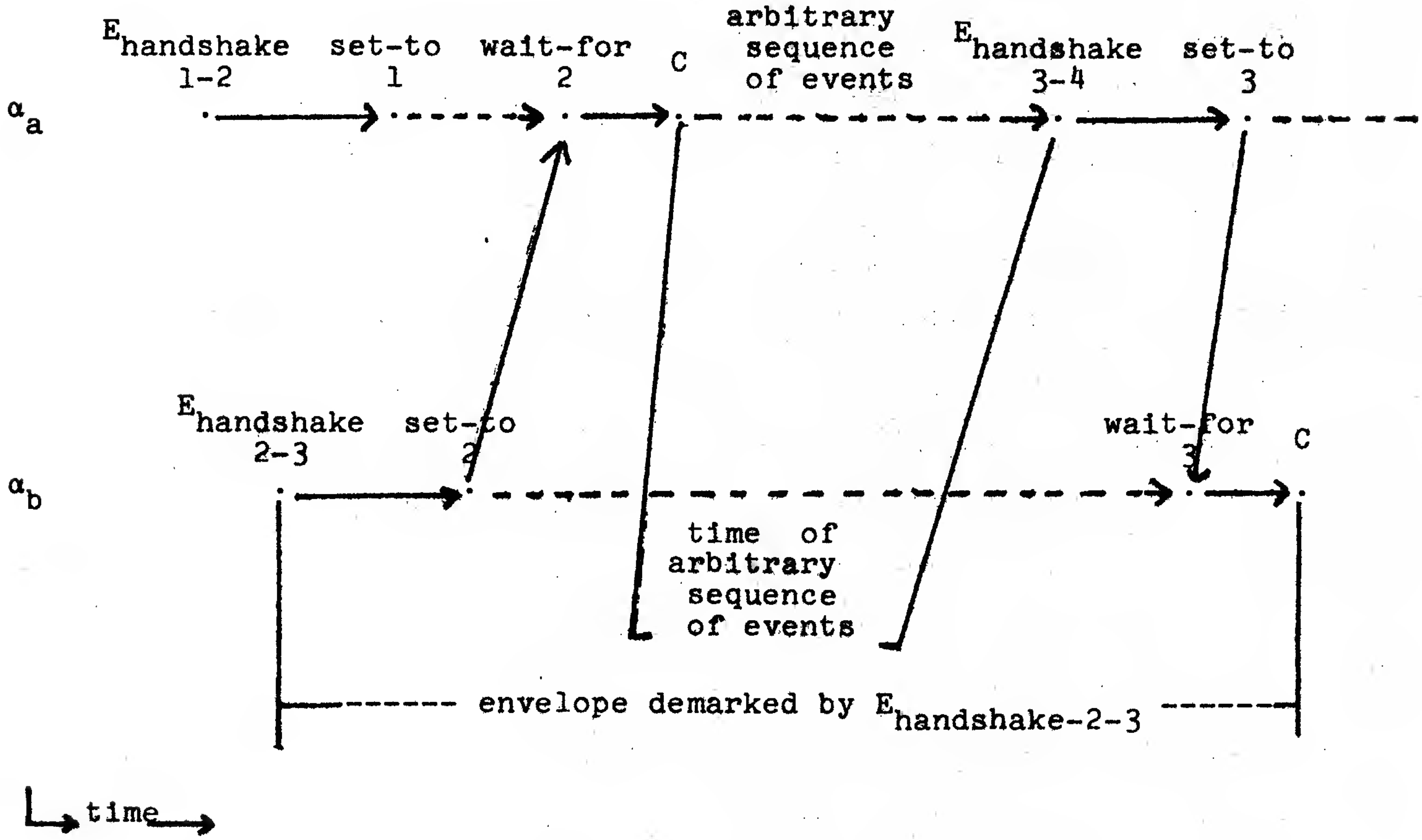
Proof: This corollary follows from successive applications of the previous theorem that states that completion causing handshakes cause completion.

In general, given a chain of completion causing handshakes occurring in activators α_a and α_b, there may be many events between the completion of one handshake in one process and the start of the next handshake in that same process. For example, suppose $\alpha_{a's}$ contribution to a handshake chain includes the events $E_{handshake-1-2}$ and $E_{handshake-3-4}$. These two events are part of a larger piece of $\alpha_{a's}$ behavior that also includes the following events and relationships:

$E_{handshake-1-2}$ before the completion of $E_{handshake-1-2}$
before an arbitrary sequence of events
before $E_{handshake-3-4}$.

Let's call the handshake event in α_b that links $E_{handshake-1-2}$ to $E_{handshake-3-4}$ in the chain, $E_{handshake-2-3}$. An important fact is that the arbitrary sequence of events in α_a noted above only occurs between $E_{handshake-2-3}$

and its completion. The handshaking causes the arbitrary events in α_a to be placed in a time envelope demarked by $E_{handshake-2-3}$ and its completion. This relationship is illustrated below:



The dashed lines represent the busywait loops present in the handshake actor.

Definition: inter-handshake gap

Let $E_{handshake-1-2}$, $E_{handshake-2-3}$, $E_{handshake-3-4}$, . . ., $E_{handshake-m-n}$, be a chain of completion causing handshakes in processes α_a and α_b. Suppose that events $E_{handshake-1-2}$, $E_{handshake-3-4}$, . . . are the events in α_a.

We will represent a general pair of successive handshake events in either process by the notation

$$E_{handshake-i-j}; E_{handshake-k-l}$$

where if we continue to use the subscript convention followed so far, i,j,k, and l are successive integers.

Let us call the completion event of $E_{handshake-i-j}$, $C_{handshake-i-j}$.

Then the sequence of events in the activator ordering of $C_{handshake-i-j}$, after that event and before the event $E_{handshake-k-l}$ is called the inter-handshake gap between j and k; that sequence will be denoted gap-j-k.

The first event of gap-j-k -- i.e. the very next event in the activator ordering following $C_{handshake-i-j}$ -- is called $E_{gap-j-k}$. The last event of the gap -- i.e. the event preceeding $E_{handshake-k-l}$ -- is called the completion event of the gap and will be written $C_{gap-j-k}$.

Definition: parallel inter-handshake gaps and handshakes

Let $E_{handshake-1-2}$, $E_{handshake-2-3}$, $E_{handshake-3-4}$, . . ., $E_{handshake-m-n}$ be a chain of completion causing handshakes. Let gap-2-3, gap-3-4, . . ., gap-(m-1)-m be the corresponding inter-handshake gaps.

Then we will say that the pairs $(E_{handshake-3-4}, gap_{-3-4})$, ..., $(E_{handshake-(m-1)-m}, gap_{-(m-1)-m})$ are each parallel inter-handshake gaps and gaps.

<u>Theorem</u>: the envelopment of inter-handshake gaps

Let $E_{handshake-1-2}$, $E_{handshake-2-3}$, ..., $E_{handshake-m-n}$ be a chain of completion causing handshakes. Let gap-2-3, gap-3-4, ..., gap-(m-1)-m be the corresponding inter-handshake gaps.

Then for any inter-handshake gap, gap-j-k, and its parallel handshake $E_{handshake-j-k}$, the following relationsips hold:

$E_{handshake-j-k}$ before $E_{gap-j-k}$

before $C_{gap-j-k}$ before $C_{handshake-j-k}$.

Of in other words, the gap is enveloped in a time period demarked by its parallel handshake.

<u>Proof</u>: It follows from the definition of inter-handshake gaps that $C_{handshake-i-j,\ i\ =\ j-1}$ -- the completion event of the handshake preceeding the gap -- is before $E_{gap-j-k}$. And similarly $C_{gap-j-k}$ is before the first event of the next handshake, $E_{handshake-h-l,\ l\ =\ k+1}$. We shall prove the theorem by proving that

$E_{handshake-j-k}$ is before $C_{handshake-i-j}$

and $E_{handshake-k-l}$ is before $C_{handshake-j-k}$

which yields the overall relationship

$E_{handshake-j-k}$ before $C_{handshake-i-j}$ before $E_{gap-j-k}$
before $C_{gap-j-k}$ before $E_{handshake-h-l}$ before $C_{handshake-j-k}$

$E_{handshake -j-k}$ is before $C_{handshake-i-j}$ because $E_{handshake-j-k}$ is a completion-causing handshake of $E_{handshake-i-j}$. Furthermore, $E_{handshake-i-j}$ may not complete before $E_{handshake-j-k}$ occurs by virtue of the "chain of completion-causing handshakes" corollary.

Similarly $E_{handshake-k-l}$ is before $C_{handshake-j-k}$ because $E_{handshake-k-l}$ is the completion-causing handshake of $E_{handshake-j-k}$.

The two results of most relevance to us here are these:

(1) If we can establish that two processes interact as a chain of completion-causing handshakes, then the inter-process interaction once begun will complete up to the last handshake. If the last handshake can be completed, also, through some separate mechanism then the entire interaction will complete.

(2) The events in one process's inter-handshake gaps will always be enclosed timewise in a handshake in the other process.

2.2.2 Reformulation of the algorithm using handshakes

In this section we will restate the algorithms that constitute the actors alias-protected-actor and regulate-mutual-exclusion replacing the explicit busywaiting loops in those algorithms by invocations of the handshake actor. We will prove that the interactions of those two actors does take the form of a chain of completion-causing handshakes and therefore the major theorems of the previous section do apply to these actors.

The reformulated algorithms are presented below. The equivalence of the original algorithms to these revised ones is apparent by even a textual substitution of the body of the handshake actor for each call to it. As usual we first present an informal description followed by a formal specification in LISP.

```
alias-protected-actor ≡  (revised)
     (1) receive argument and call it the-input-message.
     (2) set local identifier i = the name of the process.
     (3) send handshake the message

           (message: (shake: state-array [i]
                     (set-to: 'request')
                     (then-wait: 'grant'))
                     (reply-to: step (4))).
     (4) send protected-actor the message
           (message:  the-input-message
                     (reply-to: step (5)).
     (5) send handshake the message
           (message: (shake: state-array [i]
                     (set-to: 'done')
                     (then-wait: 'idle'))
                     (reply-to: step (6))).
     (6) exit to external continuation with the answer received from
protected-actor in step (4).
     END

regulate-mutual-exclusion ≡ (revised)
     (1) set cell i:= first process name known.
     (2) ask state-array[i] for its contents and let state = contents.
     (3) thre are two cases for state:
          (3-1) state = 'idle' -- go to end of loop, step (6), to
                continue scan's
          (3-2) state = 'request' -- proceed with step (4).
     (4) send handshake the message
          (message: (shake: state-array [i]
                     (set-to: 'grant')
                     (then-wait: 'done'))
                     (reply-to: step (5))).
     (5) update state-array [i]:= 'idle'.
     (6) resume or continue scanning the state-array as follows:
     (7) there are two cases for the process name index, i:
          (7-1) i = last process name known -- update i:= first process
                name known;
          (7-2) else, update i:= next process name after i.
     (8) repeat from step (2).
     END
```

```
(defun alias-protected-actor (the-input-message) ; revised
      (prog (i answer)
Step-2      (setq i (the-process-name)
Step-3      (handshake state-array[i] 'request' 'grant')
Step-4      (setq answer (protected-actor  the-input-message))
Step-5      (handshake state-array[i] 'done' 'idle')
Step-6       (return answer)))
(defun regulate-mutual-exclusion nil     ;revised
            (prog (i state)
Step-1      (setq i (first-process-name-known))
Step-2      (setq state state-array[i])
Step-3      (cond
Step-3-1    ((equal state 'idle') (goto step-6))
Step-3-2    ((equal state 'request') (goto step-4))
            (else (error))
Step-4      (handshake state-array[i] 'grant' 'done')
Step-5      (set state-array[i] 'idle')
Step-6      ;; resume or continue scanning the state-array
Step-7      (cond
Step-7-1    ((equal i (last-process-name-known))
                      (setq i (first-process-name-known)))
Step-7-2    (else (setq i (next-process-name-after i))))
Step-8      (goto step-2)))
```

In order to use our actor/behavioral proof techniques to prove properties of these programs we must first establish some corelation between steps in the algorithms and events in the behavior of the system that results when the programs are executed. This corelation is somewhat complicated by the repetitive character of the programs; we may expect that a process will attempt to enter the protected-actor many times during its execution and so each program step will be executed many times even for one particular process.

Each program step generates a class of events all of very similar form. For example each time the alias-protected-actor receives a message -- step (1) of the algorithm -- an event of the form

```
<alias-protected-actor receives
     (message: the-input-message
               (reply-to: continuation)) in $\alpha_i$>
```

occurs. We will find it useful to classify events both by the program step to which they correspond and by the value of the index i applicable to that step. Thus the above event might be classified as "an event resulting from step (1) in alias-protected-actor with index = i". We shall adopt a more succinct and mneumonic nomenclature along those lines. Events corelated with steps in alias-protected-actor will be denoted by $E_{\text{subscript-i}}$, where the subscript will have some mneumonic appeal; events resulting from regulate-mutual-exclusion will be written $M_{\text{subscript-i}}$ (the "M" is to remind us that regulate-mutual-exclusion is run by $\alpha_{\underline{\text{m}}\text{utex}}$ only).

We remind the reader that appelations such as $E_{subscript-i}$ name classes of similar events. But where no confusion is likely to result we shall use the same symbol to refer to specific events; when we must refer to more than one event of a class, we will differentiate the names with primes, e.g. $E'_{subscript-i}$.

Definition: names of events resulting from the algorithms

*Let $E_{hello-i}$ denote the events whereby alias-protected actor receives an input message:

$E_{hello-j}$: <alias-protected-actor receives
(message: the-input-message
(reply-to: continuation)) in α_i>.

*Let $E_{request-i}$ be the handshake event at step (3) of the actor:

$E_{request-i}$: <handshake receives
(message: (shake: state-array[i]
(set-to: 'request')
(then-wait: 'grant'))
(reply-to: step (4))) in α_i>.

*Let $E_{enter-i}$ name the events whereby the-input-message is relayed to the protected-actor. $E_{enter-i}$ is the first event that references the critical protected-actor

$E_{enter-i}$: <protected-actor receives
(message: the-input-message
(reply-to: step (5))) in α_i>.

*Let E_{exit-i} denote events of protected-actor replying to the alias.

E_{exit-i}:<step (5) of alias-protected-actor receives
(message: the-answer) in α_i>.

*Let E_{done-i} be the handshake event at step (5):

E_{done-i}: <handshake receives
(message: (shake: state-array[i]
(set-to: 'done')
(then-wait: 'idle'))
(reply-to: step (6))) in α_i>.

*And let $E_{byebye-i}$ denote the events whereby alias-protected-actor transmits protected-actor's answer to the outside world:

$E_{byebye-i}$: <continuation receives
(message: the-answer) in α_i>.

Now we define events in regulate-mutual-exclusion.

*Let M_{scan-i} be the event in step (2) that scans state-array[i].

M_{scan-i}: <state-array[i] receives
(message: ['contents?']
(reply-to: step (2))) in α_{mutex}>.

*The event following M_{scan-i} takes two possible forms. Let
$M_{not-request-i}$ be the response

$M_{not-request-i}$: <step (2) of regulate-mutual-exclusion receives
(message: 'idle') in α_{mutex} >;

let $M_{request-i}$ be the other response

```
    M_request-i: <step (2) of regulate-mutual-exclusion receives
                       (message: 'request') in α_mutex>.
```

*Let $M_{grant-i}$ be the handshake event at step (4):

```
    M-grant-i: <handshake receives
                      (message: (shake: state-array[i]
                                 (set-to: 'grant')
                                 (then-wait: 'done'))
                                 (reply-to: step (5))) in α_mutex>.
```

*Let M_{idle-i} denote events at step (5) that reset the element to 'idle'.

```
    M_idle-i: <state-array[i] receives
                       (message: ['update' to 'idle']
                                 (reply-to: step (6))) in α_mutex>.
```

A significant ordering relationship among the named events of alias-protected-actor may be inferred from the straight-line, non-branching nature of the actor's algorithm. Since there are no branches in the algorithm at the level of abstraction represented by the revised version here, if all the named events occur, they must occur in order.

Theorem: straight line theorem for alias-protected-actor

Given the event class names defined above.

We may structure the event classes in an ordered sequence:

$[E_{hello-i}, E_{request-i}, E_{enter}, E_{exit-i}, E_{done-i}, E_{byebye-i}]$.

If an event from any class appears in an actor system's behavior then an event from each of the preceeding classes in the sequence must also appear, in the same order as the events appear in the sequence. For example, if an event $E_{byebye-i}$ appears in a system's behavior, then the following events must also appear, before $E_{byebye-i}$, in the stated order:

$E_{hello-i}$ before $E_{request-i}$ before $E_{enter-i}$ before E_{exit-i} before E_{done-i}.

Furthermore, if two events of the same named class, call them $E_{class-i}$ and $E'_{class-i}$ appear in the behavior, then an event of each other named class must appear between $E_{class-i}$ and $E'_{class-i}$.

Proof: The order used for the sequence of events is derived by inspection of the stated text of the algorithm used by alias-protected-actor.

The inviolability of the order expressed in the theorem follows from the absence of loops and branches in the algorithm for alias-protected-actor at the level of abstraction pertaining to the named events.

A similar theorem may be derived for the events of regulate-mutual-exclusion, or at least the wait mode of the algorithm, steps (2) through (5). With alias-protected-actor the theorem explained that the events pertaining to some particular process, α_i, happen serially in a well-defined sequence. Regulate-mutual-exclusion differs in that it interacts with all the external processes, not just one particular α_i. But since only one process ever runs the actor, the events therein pertaining to all processes will occur serially. This is the property that is fundamentally depended on for the algorithm to provide mutual exclusion.

Theorem: straight line theorem for regulate-mutual-exclusion

Given the event class names defined above.

And given the assumption that α_{mutex} is the only process that ever executes the actor regulate-mutual-exclusion.

We may structure the following event classes in an ordered sequence:

$[M_{request-i}, M_{grant-i}, M_{idle-i}]$.

If an event from any of the classes appears in a system's behavior then an event from each of the preceeding classes in the sequence must also appear, in the same order as the events appear in the sequence.

Furthermore, given two events $E_{class-i}$ and $E_{class-j}$, where the word "class" may be instantiated by one of "request", "grant", or "idle" -- if $E_{class-i}$ and $E_{class-j}$ both appear in the behavior, then at least the following other events must appear in order:

(1) the events after $E_{class-i}$ in the sequence;

(2) the events before $E_{class-j}$ in the sequence.

For example, if the events $M_{grant-a}$ and $M_{grant-z}$ both occur then events M_{idle-a} and $M_{request-z}$ must also occur in that order.

Proof: The fact that only one process ever executes regulate-mutual-exclusion means that the behavior pertaining to it is a total order and the order of events may be read off the text of the actor's algorithm.

The order is fixed and inviolable due to the absence of loops and branches in that part of the algorithm giving use to $M_{request-i}$, $M_{grant-i}$, and M_{idle-i}.

As a final result of this section we will prove that the two actors interact as a chain of completion causing handshakes. There are three handshake events involved in any interaction between alias-protected-actor and regulate-mutual-exclusion -- they are $E_{request-i}$, E_{done-i}, and $M_{grant-i}$. The chain that is formed in $E_{request-i}$, $M_{grant-i}$, E_{done-i}. We will prove that as used in these programs $M_{grant-i}$ always causes the completion of $E_{request-i}$ and E_{done-i} always causes the completion of $M_{grant-i}$. Thus the three events form a chain of completion-causing handshakes.

Theorem: $M_{grant-i}$ causes completion of $E_{request-i}$.

Given the initial conditions stated for the main theorem, in particular the fact that state-array[i] is initialized to 'idle'.

Then $M_{grant-i}$ is always a completion-causing handshake of $E_{request-i}$.

Proof: From the definition, $M_{grant-i}$ is a completion causing handshake of $E_{request-i}$ if and only if

(1) $M_{grant-i}$ is a matching handshake for $E_{request-i}$;

(2) $M_{grant-i}$ occurs after the event $E_{set-to-request-i}$, the event within the handshake that updates the cell to 'request'; but before the completion of $E_{request-i}$;

and (3) there are no events $E_{clobber}$ between $E_{set-to-request-i}$ and the completion of $E_{request-i}$, where $E_{clobber}$ is an event of the form

$E_{clobber}$:< state-array[i] receives
(message: ['update' to z]
(reply-to: ?)) in α_z >, z $\neq$ 'grant'.

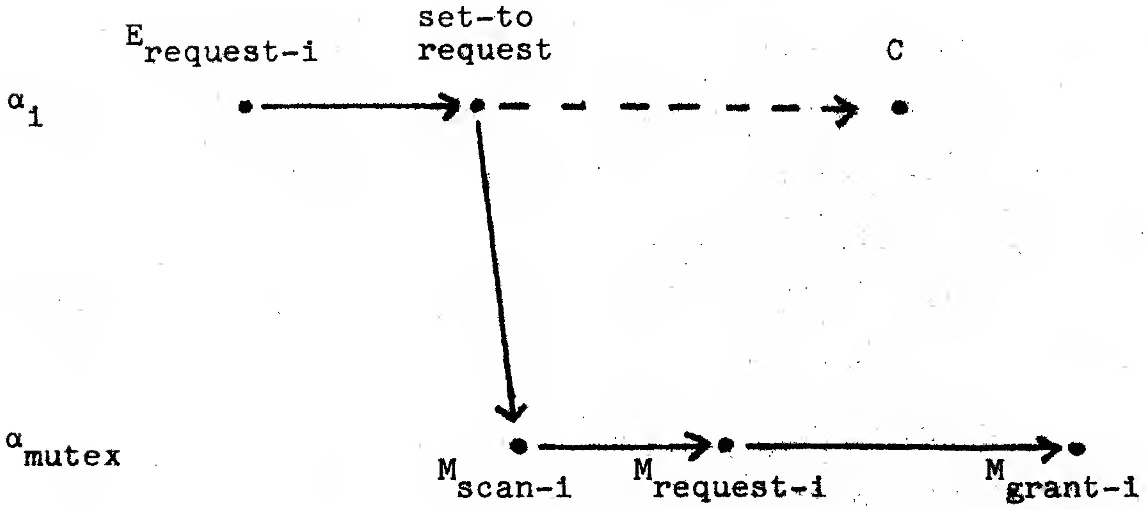
The matching criterion is purely syntactic. Two handshakes match if the "then-wait:" of one equals the "set-to:" of the other. $E_{request-i}$ waits for state-array[i] = 'grant' and $M_{grant-i}$ sets state-array[i] equal to 'grant'; hence $M_{grant-i}$ matches $E_{request-i}$.

The other two criteria are dependent on how the handshakes are used in a specific program. Our theorem states that $M_{grant-i}$ must always be a completion-causing handshake of $E_{request-i}$ and so we must prove that in all executions of the actor system the proper orderings will hold.

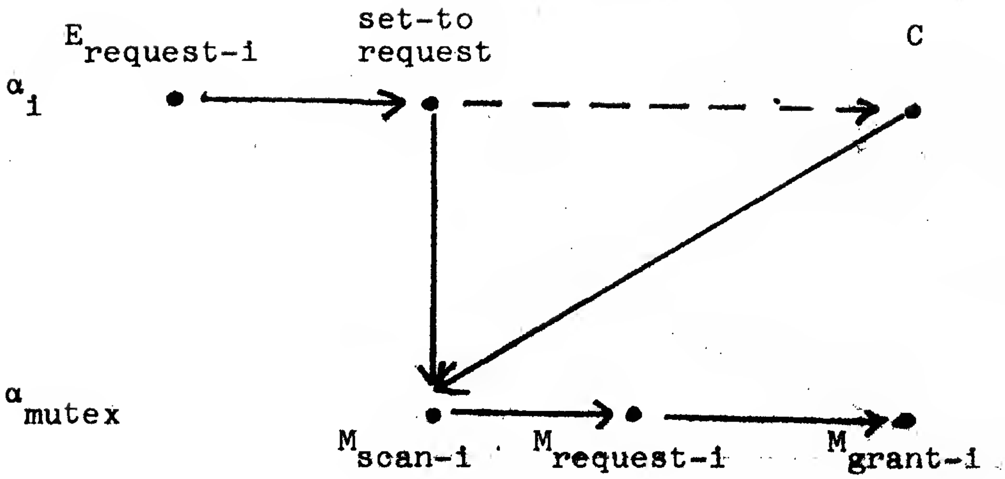
By the straight line theorem if $M_{grant-i}$ appears in the behavior it must be preceeded by an event $M_{request-i}$. $M_{request-i}$ is an event whereby state-array[i] reports that its contents equal 'request'. The scan event, M_{scan-i}, that sends the 'contents?' message to state-array[i] must occur while state-array[i] equals 'request'.

Since state-array[i] is initialized to 'idle', M_{scan-i} in this situation must occur after an update event that updates the cell to 'request', and before any event that resets state-array[i] to any other value. A perusal of the algorithms here indicates that the only events that update state-array[i] to 'request' are the "set-to:" events in $E_{request-i}$. So the M_{scan-i} giving rise to an $M_{request-i}$ event may only occur after the "set-to:" event in $E_{request-i}$ and before the state is further updated.

The ordering information garnered so far is illustrated below:



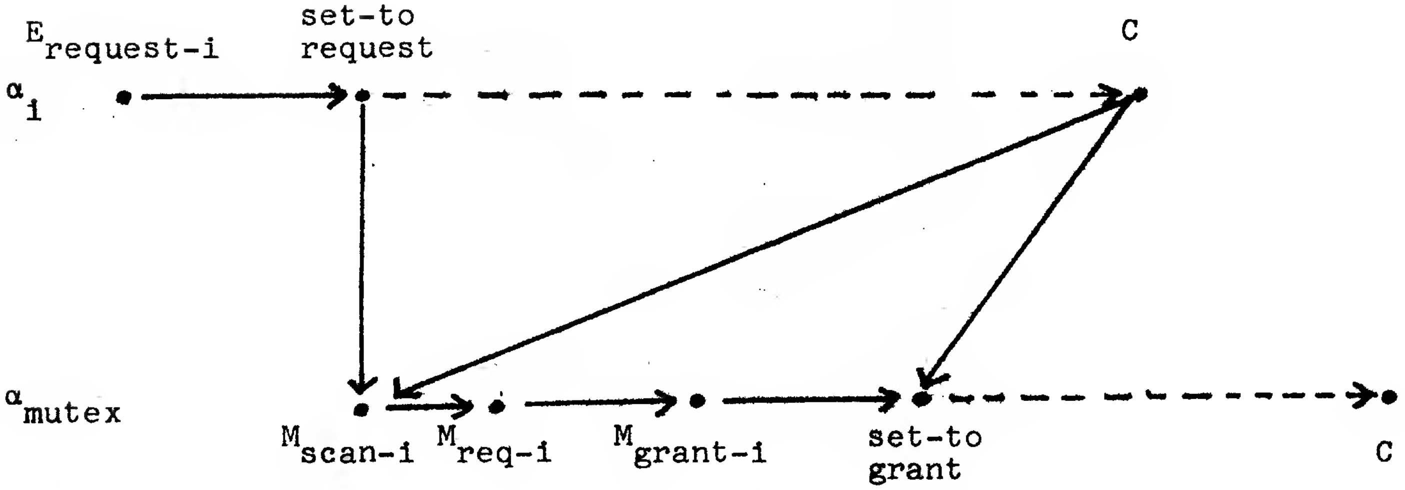
Now, we know that state-array[i] is not updated in a handshake after the "set-to:" event; and once $E_{request-i}$ completes the value of the cell will no longer be 'request' and will not be reset to 'request' until another event of the $E_{request-i}$ class occurs. Therefore, the M_{scan-i} leading to the $M_{request-i}$ event may definitely be placed in time between the "set-to:" and the completion of an $E_{request-i}$ type event:



Furthermore the handshake event will not be completed until a completion-causing event of the form

<state-array[i] receives
(message: ['update' to 'grant']
(reply-to: ?)) in $\alpha_?$ >

occurs. The only events of that form in the actor system are the "set-to:" events within the handshake $M_{grant-i}$. We have therefore placed the time of $M_{grant-i}$ and its "set-to:" event as before the completion of $E_{request-i}$ also:



So we see that $M_{grant-i}$ must always occur between an event of the form $E_{set-to-request-i}$ -- the "set-to:" event of a handshake, $E_{request-i}$ -- and the completion of the handshake. $M_{grant-i}$ thus always satisfies criterion 2 of the definition of completion-causing handshakes.

The final point that we must establish is that no event, $E_{clobber}$, updating state-array[i] to value other than 'grant' may occur between $E_{set-to-request-i}$ and the completion of $E_{request-i}$. There are three cases to consider:

(1) that $E_{clobber}$ is an event in a process other than α_i or α_{mutex};

(2) that $E_{clobber}$ occurs in α_i;

or (3) that $E_{clobber}$ happens in α_{mutex}.

Case (1) may be rejected due to our supposition that only the actors alias-protected-actor and regulate-mutual-exclusion are able to access the state-array; alias-protected-actor makes it impossible for any process but α_i to reference state-array[i]; and by assumption α_{mutex} is the only process that ever runs regulate-mutual-exclusion.

We have already noted that there are no update events in a handshake after the "set-to:" event. Therefore there are no update events in α_i between $E_{set-to-request-i}$ and the completion of $E_{request-i}$.

As for α_{mutex} we have already proved that there are no events updating state-array[i] between $E_{request-i}$ and $M_{grant-i}$. After the occurrence of the event $M_{grant-i}$ its "set-to:" event happens; this event updates the cell but it updates it to the allowed value, 'grant'. Between the "set-to:" event and the completion of $M_{grant-i}$ there are no events updating state-array[i] so if $E_{clobber}$ occurs in α_{mutex} it must happen after the completion of $M_{grant-i}$.

But $M_{grant-i}$ cannot complete until state-array[i] is updated. Since that update does not happen in α_{mutex} it must happen in α_i. But we've just shown that α_i cannot update state-array[i] further until $E_{request-i}$ completes. So α_{mutex} cannot clobber the state until α_i updates it and α_i cannot update it until $E_{request-i}$ completes. I.e. there can be no event $E_{clobber}$ in α_{mutex} prior to the completion of $E_{request-i}$, and thus case (3) is rejected also.

Having established that there can never be an event that clobbers the state-array element before $E_{request-i}$ is completed we have proved that the third and final criterion required for $M_{grant-i}$ to be a completion-causing handshake of $E_{request-i}$ is always satisfied. This completes the proof of the theorem.

Theorem: E_{done-i} causes completion of $M_{grant-i}$

E_{done-i} is always a completion-causing handshake of $M_{grant-i}$.

Proof: The proof of this theorem is similar to the proof of the previous theorem.

Theorem: the handshakes form a chain of completion-causing handshakes.

Given the actor system described in this section and the stated initial conditions.

Then the sequence of events $E_{request-i}$, $M_{grant-i}$, E_{done-i} always forms a chain of completion-causing handshakes.

Proof: The previous two theorems stated that

(1) $M_{grant-i}$ is always a completion-causing handshake of $E_{request-i}$ and (2) E_{done-i} is a completion-causing handshake of $M_{grant-i}$. The definition says that whenever these two relations hold then the events form a chain. Since the relations always hold in this actor system, the three events always form a chain of completion-causing handshakes.

2.2.3 Proving the theorem using handshakes

In the previous section we proved that the interaction between alias-protected-actor and regulate-mutual-exclusion always takes the form of a chain of completion causing handshakes. We shall use that result here to prove the main theorem of this chapter: that given the specified actor system and the specified initial conditions alias-protected-actor is a fair mutual exclusion actor for protected-actor.

The proof proceeds in two parts. The first part shows that all references to protected-actor that are funneled through alias-protected-actor occur in a mutually exclusive fashion. The second step is to prove that alias-protected-actor provides fair encasements of protected-actor. Combining these two points yields the overall theorem.

We will see that Part I of the proof follows straightforwardly from the handshake theorems developed in section 2.2.1. Part II has two sub-parts one of which also follows from the handshake theorem; the second sub-part, though, involves proving the fairness of regulate-mutual-exclusion's scan algorithm. It is in this last part that the potential for more processes being added to the system must be accounted for. This last section does not follow from the handshake theorem.

Part I -- Given the actor system and initial conditions stipulated in the main theorem.

Then all references to protected-actor that occur within alias-protected-actor occur in a mutually exclusive fashion.

Proof: The definition of mutual-exclusion says the following:

Let $E_{enter-i}$ be an event of the form

$E_{enter-i}$: <protected-actor receives
(message: any-message
(reply-to: continuation)) in α_i>

and let E_{exit-i} be the next event thereafter of the form

E_{exit-i}: <continuation receives
(message: any-answer) in α_i>.

Then for all quadruples of events

$(E_{enter-i}, E_{exit-i}, E_{enter-j}, E_{exit-j})$, $i = j$

one of the following two orders must hold:

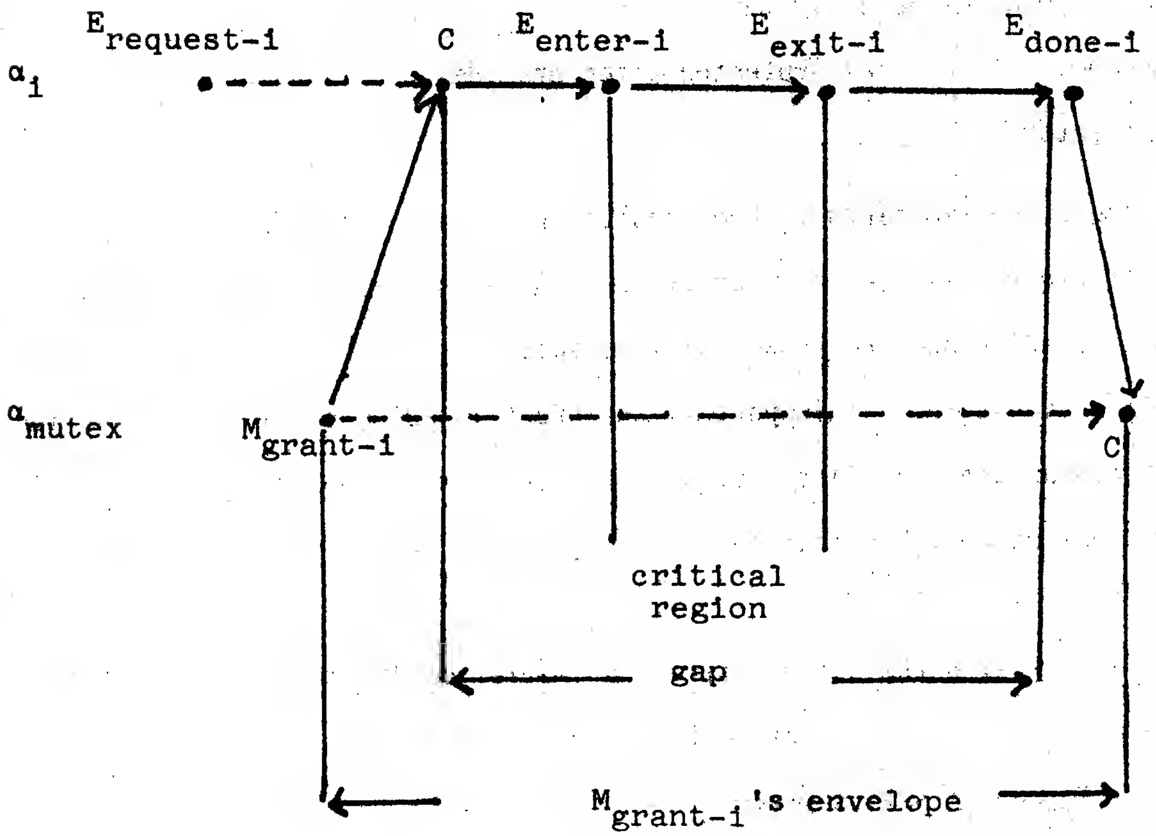
either (1) $E_{enter-i}$ before E_{exit-i} before $E_{enter-j}$ before E_{exit-j}

or (2) $E_{enter-j}$ before E_{exit-J} before $E_{enter-i}$ before E_{exit-i}.

Given a pair of events -- $E_{enter-i}$ and E_{exit-i} -- arising from the execution of alias-protected-actor, the straight line theorems for that actor shows that the events must occur between an event $E_{request-i}$ and the next event of the class E_{done-i}. Moreover $E_{enter-i}$ and E_{exit-i} must occur after the completion of $E_{request-i}$, which is a handshake event. This means that $E_{enter-i}$ and E_{exit-i} occur in the inter-handshake gap between $E_{request-i}$ and E_{done-i}.

We proved in the previous section that the sequence of handshakes $E_{request-i}$, $M_{grant-i}$, E_{done-i} is a chain of completion-causing handshakes.

It follows that $M_{grant-i}$ is the parallel handshake to the gap between $E_{request-i}$ and E_{done-i}. Therefore, from the envelopment of inter-handshake gaps theorem, we know that $M_{grant-i}$ "envelopes" the gap -- i.e. $M_{grant-i}$ is before the completion of $E_{request-i}$, and the completion of $M_{grant-i}$ is after E_{done-i}. And since $E_{enter-i}$ and E_{exit-i} are within the gap, $M_{grant-i}$ envelopes them too. See the illustration below:



The straight-line theorem for the actor regulate-mutual-exclusion shows that if an event $M_{grant-i}$ appears in the behavior, then no other event of the class $M_{grant-j}$ may appear until the completion of $M_{grant-i}$ appears. This leads to the following relationship:

$M_{grant-i}$ before $E_{enter-i}$ before E_{exit-i} before completion of $M_{grant-i}$ before $M_{grant-j}$ before $E_{enter-j}$ before E_{exit-j}, for all values of i and j.

Embedded in that relationship is the relationship required by the hypothesis, and thus Part I of the theorem is proved.

Part II -- Given the actor system and initial conditions stipulated in the main theorem. Then alias-protected-actor provides fair encasement for protected-actor.

Proof: There are two parts to the proof. First we will show that if a particular request is scanned then the associated message will be transmitted to protected-actor and the answer will subsequently be retransmitted to the external continuation. This part of the proof is essentially a continuation of the previous proof of Part I of the main theorem. The second section of this proof establishes that the scanning of the array is itself fair.

Part II-(a) -- Given the actor system and initial conditions stipulated in the main theorem. If an event of the form

$E_{hello-i}$: <alias-protected-actor receives
(message: any-message
(reply-to: continuation)) in α_i>

appears in the behavior, and it is followed by an event of the form

$M_{request-i}$: step (2) of regulate-mutual-exclusion receives

(message: 'request') in α_{mutex} ,

then the following events also will appear in the behavior after $M_{request-i}$ and in the stated order:

$M_{enter-i}$: <protected-actor receives

(message: any-message

(reply-to: step (8))) in α_i>

before

E_{exit-i}: <step (8) of alias-protected-actor receives

(message: any-answer) in α_i>

before

$E_{byebye-i}$:<continuation receives

(message: any-answer) in α_i>.

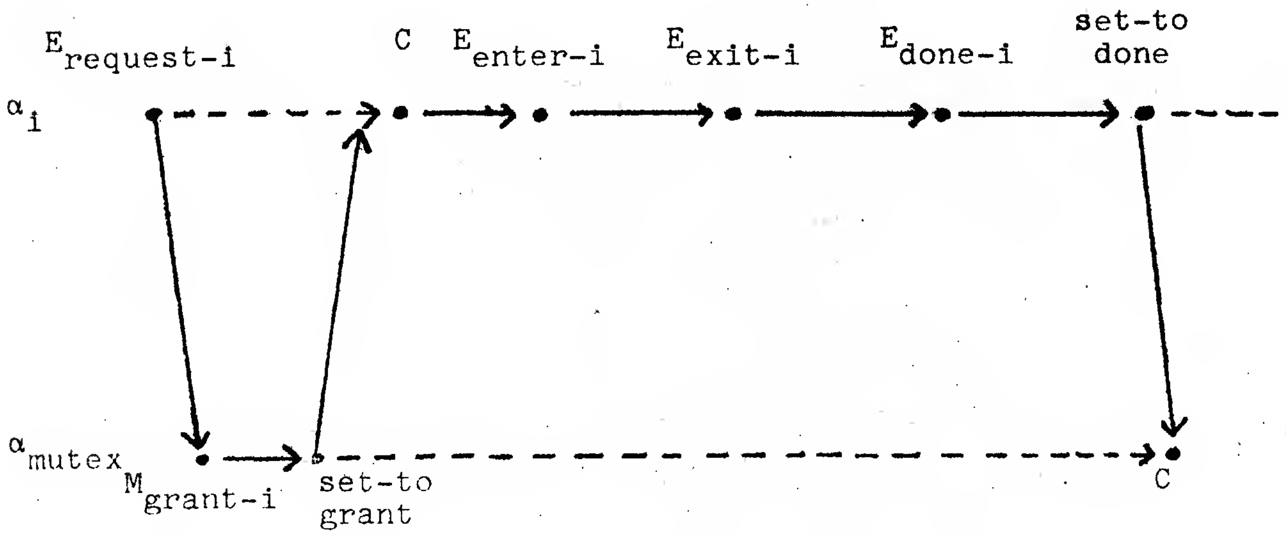
<u>Proof of Part II-(a):</u> Certain parts of the hypothesis are true by inspection and are included here for completeness. In particular, if $E_{enter-i}$ occurs, then E_{exit-i} is required to occur because protected-actor is constrained to be a normal returning actor in the hypothesis of the main theorem.

If $E_{hello-i}$ occurs then by virtue of the straight line theorem the next named event, $E_{request-i}$, is bound to happen. Also if $M_{request-i}$ occurs that means that $M_{grant-i}$ will occur, too.

We proved in the previous section that the sequence of events $E_{request-i}$ and $M_{grant-i}$, E_{done-i} form a completion causing chain of handshakes and so if $E_{request-i}$, and $M_{grant-i}$ both occur, then E_{done-i} will occur inevitably after. But since E_{done-i} is the last handshake in the chain it will not become completed by virtue of activities in the chain; E_{done-i} requires an explicit completion-causing event in order for that handshake to terminate and for the next step in alias-protected-actor to occur. By the same token if E_{done-i} completes then the next step of the program will happen, too; the step after the handshake is the step that replies to the external continuation -- i.e. it is the step corresponding to $E_{byebye-i}$.

So if there will be a completion-causing event for E_{done-i} during every execution the system then the theorem is proved.

The state of affairs in effect when E_{done-i} occurs is illustrated below. We know that $M_{grant-i}$ always completes after the "set-to:" in E_{done-i} because E_{done-i} is always a completion-causing handshake of $M_{grant-i}$:



The event in α_{mutex} after the completion of $M_{grant-i}$ is M_{idle-i}:

M_{idle-i}: <state-array[i] receives
(message: ['update' to 'idle']
(reply-to: step (6))) in α_{mutex}> .

We will prove that M_{idle-i} is always the one and only completion-causing event of E_{done-i}.

M_{idle-i} must satisfy three criteria in order to completion-causing event of E_{done-i}:

(1) M_{idle-i} must match E_{done-i} -- which it does by inspection;

(2) M_{idle-i} must occur after the "set-to:" event related to E_{done-i} -- which it does because E_{done-i} is always a completion-causing handshake of $M_{grant-i}$ which is before M_{idle-i};

and (3) there must be no other event updating state-array[i] between E_{done-i}'s "set-to:" event and the completion of E_{done-i}. This statement of the third criterion is stricter than the definition requires; if this statement proves to be true then M_{idle-i} will always be the one and only completion-causing event of the handshake.

The third criterion is true as follows: The only processes that may reference state-array[i] are α_i and α_{mutex}. There is no update event between the "set-to:" event of a handshake and its completion and so no event in α_i may violate the criterion. There are no more events in α_{mutex} that update state-array[i] after M_{idle-i} during this same cycle of the algorithm. If α_{mutex} updates state-array[i] further, that update may not occur until after an event $M'_{request-i}$. We have already proved that events of the $M_{request-i}$ class occur only during an $E_{request-i}$ class handshake and therefore there must be an event $E'_{request-i}$ between M_{idle-i} and $M'_{request-i}$; also if the alleged update event is to happen before the completion of E_{done-i}, then $E'_{request-i}$ must likewise occur before the completion of E_{done-i}. In other words the supposition that we are making is that there is an event $E'_{request-i}$ between E_{done-i} and its completion.

The existence of such an event in that relationship leads to the vollowing ordering:

E_{done-i} before $E'_{request-i}$ before completion of E_{done-i} before $E_{byebye-i}$. This ordering of events violates the straight line theorem of alias-protected-actor. Hence there cannot be an event of the form $E'_{request-i}$ between E_{done-i} and its completion and the third criteria is upheld.

So M_{idle-i} will always cause the completion of E_{done-i}; the event $E_{byebye-i}$ is assured of occuring, and Part II-(a) of the theorem is proved.

Now we must establish that the scanning of the array is itself done fairly. Part II-(a) showed that any request that does get scanned will get passed through to the protected-actor. Part II-(b) shows that all requests that are made will be scanned.

<u>Part II-(b)</u> -- Given the actor system and the initial conditions stipulated in the main theorem. If an event of the form

$E_{hello-i}$: <alias-protected-actor receives
(Message: any-message
(reply-to: continuation)) in α_i>

appears in the behavior, then it will be followed by an event of the form

$M_{request-i}$: <step (2) of regulate-mutual-exclusion receives
(message: 'request') in α_{mutex}>.

Also, if $E_{hello-i}$ happens and is followed by $M_{request-i}$, then there will be no subsequent event $M'_{request-i}$ like $M_{request-i}$, unless there is an event $E'_{hello-i}$ after $E_{hello-i}$.

Proof of Part II-(b): The second statement in the hypothesis is quite easy to establish and we will dispose of it first. What the statement is saying is that if a request is scanned once it will not be scanned again. In other words the operation of letting a request into the protected-actor and passing the answer back out reinitializes the state-array element somehow.

We have already shown that if an event like $M_{request-i}$ occurs it must be immediately preceeded by an event M_{scan-i} that reads the contents of state-array[i]. $M_{request-i}$ follows only if the contents of the state element were 'request'. If $M_{request-i}$ does happen then M_{idle-i} is assured of occurring and there cannot be another M_{scan-i} type event in between.

M_{idle-i}, of course, sets state-array[i] to be 'idle'. There cannot be another event of the $M_{request-i}$ class until and unless state-array[i] is again made equal to 'request'. The only events in the system that do that are

$E_{request-i}$ type events; $E_{request-i}$ events in turn only occur after $E_{hello-i's}$, and only one $E_{request-i}$ will follow any $E_{hello-i}$ without an intervening $E_{hello-i}$.

So, what we have is this:

in α,: $E_{hello-i}$ before $E_{request-i}$ before $E'_{hello-i}$ before $E'_{request-i}$
in α_{mutex}: $M_{request-i}$ before M_{idle-i} before $M'_{request-i}$

which says that if $E_{request-i}$ is before $M_{request-i}$ then $E'_{request-i}$ must be before $M'_{request-i}$ -- i.e. the second statement is established.

The first statement in the hypothesis is stating that the scan of the array must be fair. Let M_{scan-i} be an event of the form

M_{scan-i}: <state-array[i] receives
(message: ['contents?']
(reply-to: step (2))) in α_{mutex}>.

The scan algorithm is fair if and only if for each process in the system there is a first scan event, M_{scan-i}, and after each M_{scan-i} event there is another one later. If those conditions hold then there will be an M_{scan-i} event after each $E_{request-i}$ event which implies that there will be an $M_{request-i}$ event after each $E_{request-i}$ event also. Since each $E_{request-i}$ event is a direct result of the preceeding $E_{hello-i}$ event, this would mean that every $E_{hello-i}$ would be followed by an $M_{request-i}$ as required by the theorem.

Events M_{scan-i} occur in step(2) of regulate-mutual-exclusion. The value of i being scanned is controlled by step(1) once, and thereafter by steps (6) and (7). Assume for the moment that the scan is not interrupted by any request, and let's examine just the scan part of the algorithm regulate-mutual-exclusion ≡ (scan part only)

```
(1) set cell i:= first process name known.
(2) ask state-array[i] for its contents and let state = contents.
...
(6) resume or continue scanning the state-array as follows:
(7) there are two cases for the process name index, i:
    (7-1) i = last process name known -- update i:= first process name known;
    (7-2) else, update i:= next process name after i.
(8) repeat from step (2)
END
```

If the state-array is static and does not change size with time then the first process name and the last process name are each constants. The scan algorithm then has the form of two nested loops: the outer loop repeatedly sets i to a constant initial value and the inner loop updates i until it reaches a constant maximum value. This structure will result in fair scanning if the function updating i has the following property: starting with the first process name known, successive application of the function must yield all process names known to the system; if the function is truly functional in the mathematical sense then it will yield all process names once before returning any name a second time.

If the scan is interrupted then some event M_{scan-i} will lead to an $M_{request-i}$ event rather than directly going to step (9). But as we have already seen, if $M_{request-i}$ happens then all the steps of the actor regulate-mutual-exclusion must inevitably ensue leading to M_{idle-i} in step (5). After step (5), the scan is resumed in step (6). Since the wait mode interruption is guaranteed to terminate and to resume the scan, and since it does not effect the scan parameter i in any way. The interruption cannot modify the fairness of the scan algorithm.

Our argument for fairness assumes that the state-array is of constant size. Let us relax that assumption and allow the array to grow without bound over time. To do so requires that the state-array be treated as a variable sized structure instead of a fixed size array. Also the index i must not be interpreted as the usual kind of integer subscript; i is instead a possibly symbolic index into the state-array structure.

The fairness argument heretofore was based on the fact that between two successive scans of some state-array element there are a fixed number of other elements to scan -- namely the size of the array minus one. To extend the arguments we must replace the notion of a fixed number of intermediate scans by the notion of a bounded number. If between two successive scans of the same state-array element there are only a bounded number of other elements to scan, then the second successive scan event always will happen and the algorithm will remain fair.

It turns out to be crucial where in the state-array structure the newly added elements are put. If they are added after the current scan point -- i.e. between the element currently being scanned and the end of the structure -- then the scan algorithm no longer is assured of being fair. For if elements are added to the array as fast as α_{mutex} is able to scan them or faster, then α_{mutex} will never reach the end of the array and will never wrap around to the beginning. Thus no element of the array will ever get scanned a "next" time.

Suppose, though, that the new elements are always added behind the scan -- i.e. between the beginning of the array and the element being scanned currently. In this case there will always be a bounded number of elements between the current scan point and the last element of the array. That is, although the beginning point of the scan algorithm may change now, or some behavior in the middle may vary, the stop rule for the algorithm remains constant. So everytime $\underline{i}$ is set to the first process name known we may predict the number of entries that must be scanned before reaching the constant, last process name. Everytime $\underline{i}$ is reset to the first process we will say that a new cycle of the algorithm has begun. Let us call the number of entries that must be scanned in some particular cycle, size (cycle).

Now suppose that an event M_{scan-i} for some particular $\underline{i}$ has just occurred. We must prove that there will be another event M'_{scan-i} in the behavior within a bounded number of scan steps. Call the cycle during which M_{scan-i} occurred cycle-a and the next cycle call cycle-a+1. The number of scan events between M_{scan-i} and M'_{scan-i} is less than

size (cycle-a) -- the total number of scan events in cycle-a

+ size (cycle-a+1) -- the total number of scan events in the next cycle.

Thus if the array is expanded behind the current scan, each event of the form M_{scan-i} will be followed by another such event within a bounded number of scan events. The bound is not known at the time that M_{scan-i} happens; it is, however, known a bounded time later, when the first process in the state-array structure is next scanned. The scan algorithm therefore remains fair even if the state-array is expanded without bound over time.

A specific algorithm for expanding the state-array:

A specific algorithm for expanding the state-array was presented in the informal discussion of the theorem. That algorithm treated the state-array as a list rather than an array. In that model the identifier "state-array" is used to name the list of state-array elements. The notation -- state-array[i] -- must be understood as a symbolically indexed reference into the list that state-array points at. That is, the expression

state-array[i]

may be assumed to return a pointer to the cell on the state-array list for the _i_th process.

The programs for manipulating state-array, alias-protected-actor and regulate-mutual-exclusion, were written with the idea in mind that state-array would indeed be an array. Now that state-array is to be a list those programs might have to be modified.

All references to state-array that occur in alias-protected-actor always refer to a specific element of it. Since the expression -- state-array[i] -- returns a pointer to the appropriate cell, the references in alias-protected-actor will continue to work even though state-array is a list.

The algorithm for regulate-mutual-exclusion though references all the entries of state-array sequentially and it is more convenient to rewrite that actor using car's and cdr's than it is to try to make the array notation work. A version of regulate-mutual-exclusion that is particularized for the case of state-array being a list is presented on the following page:

```
(defun regulate-mutual-exclusion nil
     (prog (i-state state')
           (setq i-state state-array)
Step-2     (setq state (car i-state))
           (cond
                ((equal state 'idle')(goto step-9))
                ((equal state 'request')(goto step 4))
                (else (error))
Step-4     (rplaca i-state 'grant')
           ;; loop waiting for state-array[i], i.e. i-state, to be 'done'
Step-6     (setq state (car i-state))
           (cond
                ((equal state 'grant')(goto step-6))
                ((equal state 'done')(goto step-8))
                (else (error))
Step-8     (rplaca i-state 'idle')
           ;; resume or continue scanning the state-array
Step-9     (cond                            ;if at end of state-array list
                ((null (cdr i-state))       ;then reset i-state to beginning
                (setq i-state state-array))
           (else (setq i-state (cdr i-state))))
                                            ;else set i-state to next entry.
           (goto step-2)))
```

The operation of creating a new process may add the new process to the state-array by cons'ing a cell for the new process onto state-array. This procedure is described in algorithms for actors fork and expand-state array below:

```
fork ≡
(1) receive argument and call it new-process
(2) do whatever has to be done in the innards of the system to create a
   new process
(3) expand-state-array for the new-process (see below)
(4) exit to externally supplied continuation
END

Expand-state-array ≡
(1) receive argument and call it new-process
(2) allocate a new cell and call it new-state. Update new state's initial
    contents to be 'idle'
(3) cons new-state onto state-array and let new-state-array = the returned
    value
(4) Store new-state in the bowels of the system in a manner associated with
    new-process
(5) Update state-array:= new-state-array
(6) Exit
END
```

The actor expand-state-array is expressed formally in LISP on the following page:

```
(defun expand-state-array (new process)
          (prog (new-state new-state-array)
Step-2        (setq new-state 'idle')
Step-3        (setq new-state-array (cons new-state state-array))
Step-4        (... store new-state in the system...)
Step-5        (setq state-array new-state-array)))
```

The scan algorithm in the revised regulate-mutual-exclusion here uses the pointer in the cell state-array as "the beginning of the scan". The expansion algorithm in expand-state-array always adds new processes before the current value in state-array. Therefore this method of adding processes to the list does not destroy the fairness of the scan algorithm.

However, if multiple processes are able to execute expand-state-array concurrently, it is apparent that harmful interactions are quite possible. Let us call the events indicated by step (3) E_{cons-i}:

E_{cons-i}: <state-array receives
(message: ['cons' new-state]
(reply-to: step (3))) in α_i>.

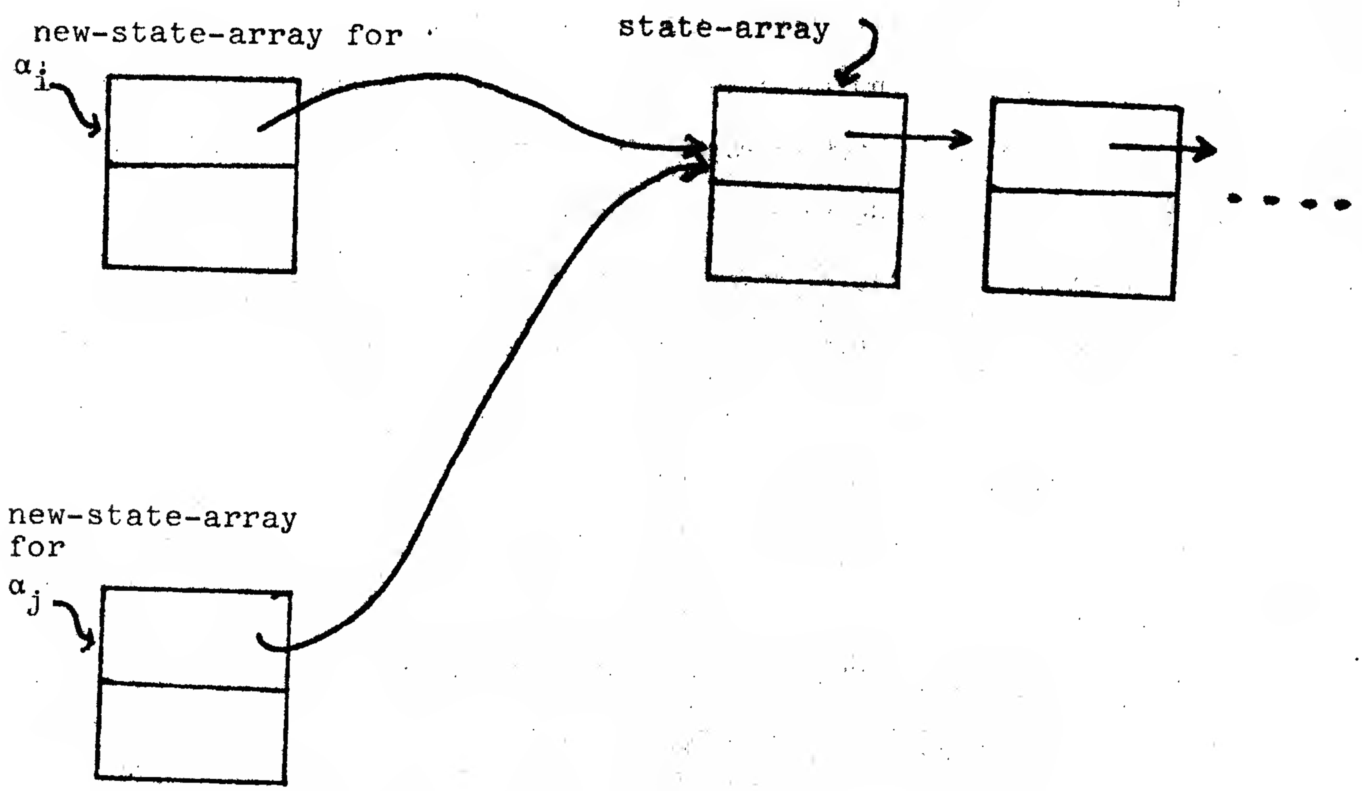
Also, let $E_{update-i}$ represent the events in step (5):

$E_{update-i}$: <state-array receives
(message: ['update' to new-state-array]
(reply-to: step (6))) in α_i>.

Suppose that two processes, α_i and α_j, are executing expand-state-array at the same time. The following ordering of events is possible:

$$\begin{tikzcd}
\text{in } \alpha_i: & E_{\text{cons-i}} \arrow[d, Rightarrow] & \text{before} & E_{\text{update-i}} \\
\text{in } \alpha_j: & E_{\text{cons-j}} \arrow[urr, Rightarrow] & \text{before} & E_{\text{update-j}}
\end{tikzcd}$$

At the time of $E_{\text{cons-j}}$, state-array has not yet been updated to include α_i's new entry. So both α_i and α_j would create the new-state-array using the same previous state-array. This leads to structures like



Whichever process updates state-array in step (5) last is the one that will win out in the end. Its new-state cell will be included in the state-array list;

the other entry, while not garbage nor possessing a dangling reference, will never be referenceable from state-array. The effect is that that process will never be noticed by regulate-mutual-exclusion; i.e. requests from that process would never be serviced. If we were to allow this situation, the mutual exclusion operator would no longer be fair.

The solution as we noted earlier is to treat expand-state-array as a protected actor and only allow it to be accessed via an encasing alias actor that provides fair mutual exclusion. If we assume that all initial processes are represented in the state-array then we may specify that process creation occurs in a mutually exclusive manner by protecting the actor fork with precisely the mutual exclusion operator that we have described here. That is we may define an actor alias-fork that is identical to alias-protected-actor, except it relays the input messages to fork instead of protected-actor. In other words we may use state-array to protect the actor that expands state-array!

The final question we must resolve is whether there can be any harmful interactions between expand-state-array and other actors that reference the state-array. Alias protected-actor only references elements of the state-array and only that one element associated with the running process. Since expand-state-array never modifies any existing components of the state-array structure there can be no harmful interactions between these two actors.

Regulate-mutual-exclusion, however, does more than reference the components of state-array, it needs to reference the start of the structure every time a new cycle begins -- i.e. every time the scan reaches the last element of the array. If expand-state-array actually succeeds at getting the newly added elements included in the scan, regulate-mutual-exclusion must use the cell state-array as the starting point of the array. That is, when the algorithm says to set i:= the first process known, that means setting i such that state-array[i] will point at the same entry that state-array points to.

Suppose that i is set to state-array concurrent with some external process executing expand-state-array. Can that lead to any harmful interactions?

If i is set after step (5) of the algorithm, then i will acquire the new value of state-array and the next element that will be scanned is newly added element. By the time step (5) happens all the processing associated with expanding the array will have already been done -- step (5) is the last activity in the algorithm except for the return. Therefore there is no difficulty with the newly added element being scanned at this time.

If i is set to state-array before step (5), i.e. between steps (1) and (5), then the value it obtains is the previous value of state-array. In this case the next element scanned will be an element added during a previous expansion of the array or an original member of the array, and will not be the element being added now. In this case the newly added member will not be scanned until a future cycle starts. We know that the element will be scanned in the future because the scanning algorithm is known to be fair.

So, the state-array structure may be expanded safely with no possible timing errors provided only that the expansion is done in a mutually exclusive fashion.

Conclusion of proof: We proved in Part I that the system of actors alias-protected-actor and regulate-mutual-exclusion combine to form a mutual exclusion operator for the actor protected-actor. In part II we proved that the system of actors fairly encase protected-actor; that is all messages that are received by the alias are retransmitted to the protected-actor, and all of the protected-actor's answers are delivered to the external continuation.

Putting together Parts I and II establishes that alias-protected-actor is a fair mutual exclusion actor for protected-actor as required by the main theorem.

3. Busywaiting Synchronization Algorithms Using Extended Cells

In the previous chapter we described an algorithm that achieves fair mutual exclusion of an arbitrary number of processes using cells as the synchronozation primitives. That algorithm requires a relatively large amount of storage though -- it requires an array of cells proportional in size to the number of processes in the system. In this chapter we will describe an algorithm that uses extended cells as the synchronization primitives; this algorithm also achieves fair mutual exclusion but requires only three extended cells of storage to do it.

An extended cell is a cell that is able to model the read-modify-write instructions that are commonplace in present-day computers. These instructions enable a process to both read and update the contents of a memory location in one indivisible activity. In other words, the mutual exclusion that is provided by the computer hardware to guard against simultaneous updates to a cell is extended ever so slightly to allow these compound instructions.

Unfair mutual exclusion is trivially achievable using read/modify/write instructions as is well known, by means of binary lock variables. Suppose we redefine cells so that they will also respond to test-and-set messages. In response to such a message, the cell will update its contents to 1, and return its previous contents to the continuation. We will see the simple unfair algorithm shortly.

Fair mutual exclusion algorithms need a wee bit more memory than just a binary lock provides and hence we need an instruction that is a little more powerful. That instruction is the *add constant* instruction. When a cell receives a message of the form

```
(message: ['add constant' 27]
          (reply-to: continuation))
```

it updates its contents to be its previous contents plus the constant, 27 in this case. The value returned to the continuation is the new updated value.

We will use the *add constant* instruction in three specific configurations only. It will be used to add 1 to a cell, to add-1 to a cell, and to test-and-set a cell. test-and-set is simulated with *add constant* instructions by means of the following algorithm:

```
test-and-set ≡
(1) add constant 1 to the cell.  Call the returned value state.
(2) there are two cases for state:
     (2-1) state ≠ 1 -- add constant - 1 to cell and return 1 as the
           value of the test-and-set.
     (2-2) state = 1 -- return 0 as the value of the test-and-set.
END
```

Hereafter we shall use the word *increment* as a synonym for *add constant* 1 and the word *decrement* for *add constant* -1. *test-and-set* will refer to the algorithm specified above.

Unfair mutual exclusion may be implemented by a simple algorithm that loops trying to set a binary lock variable:

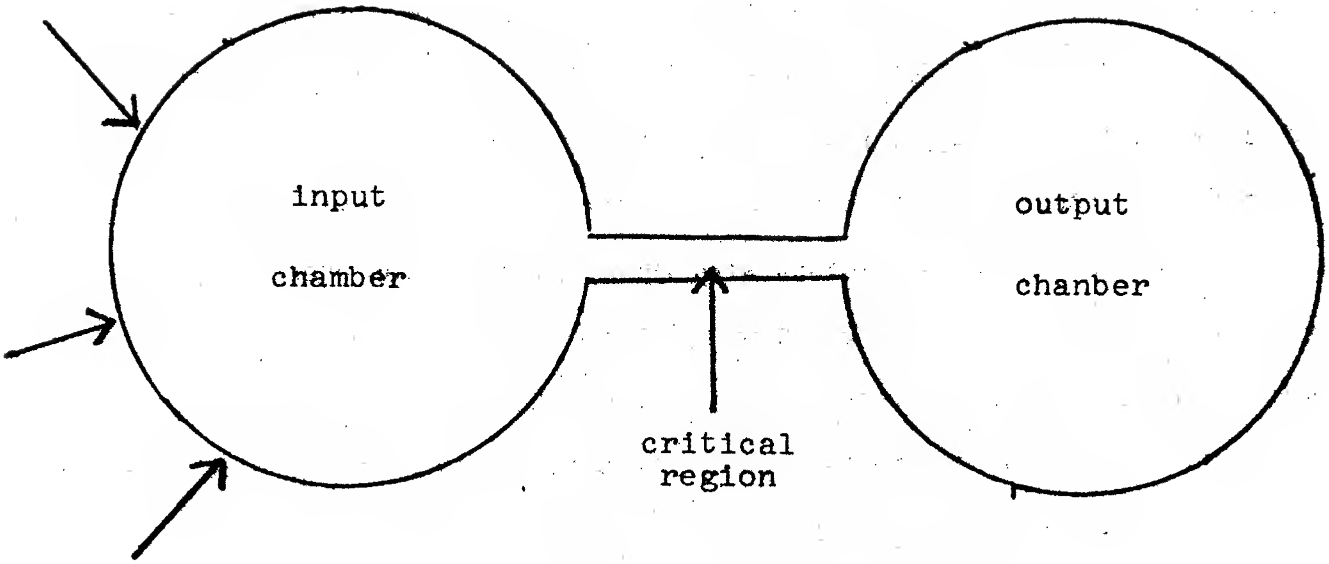
```
unfair-mutual-exclusion ≡
(1) test-and-set the cell, lock-cell, and let state = the returned
    previous contents of lock-cell
(2) there are two cases for state:
     (2-1) state ≠ 0 -- repeat from step (1)
     (2-2) state = 0 -- proceed with step (3)
(3) reference the protected critical region as required
(4) update lock-cell:=0
END
```

This algorithm is not fair for a very specific reason. Let's define the word "session" to refer to those activities occurring in some particular process, from the first time the process executes step (1) above until the process is let into the critical region once and comes out of it. I.e. a session corresponds to one interaction of a process with the critical region. A session may last forever only because during one session of some process, α_a, an unbounded number of sessions for other processes may take place.

However, at any given time there are only a bounded number of processes existant in the computation system. So how does poor α_a manage to get stuck in an infinite loop? There are two ways: either some processes are undertaking unbounded numbers of sessions -- e.g. they are in a loop wherein they enter and exit the critical region; or some processes are off somewhere busily creating new processes in an unbounded fashion, and these newly created processes are engaging in sessions with the critical region over here. If both these "session sources" could be muted so that during one session for $_a$ only a bounded number of other sessions could take place, then α_a could not get stuck. And indeed both sources can be quenched by a scheme that requires only three read/modify/write type cells, as we shall show.

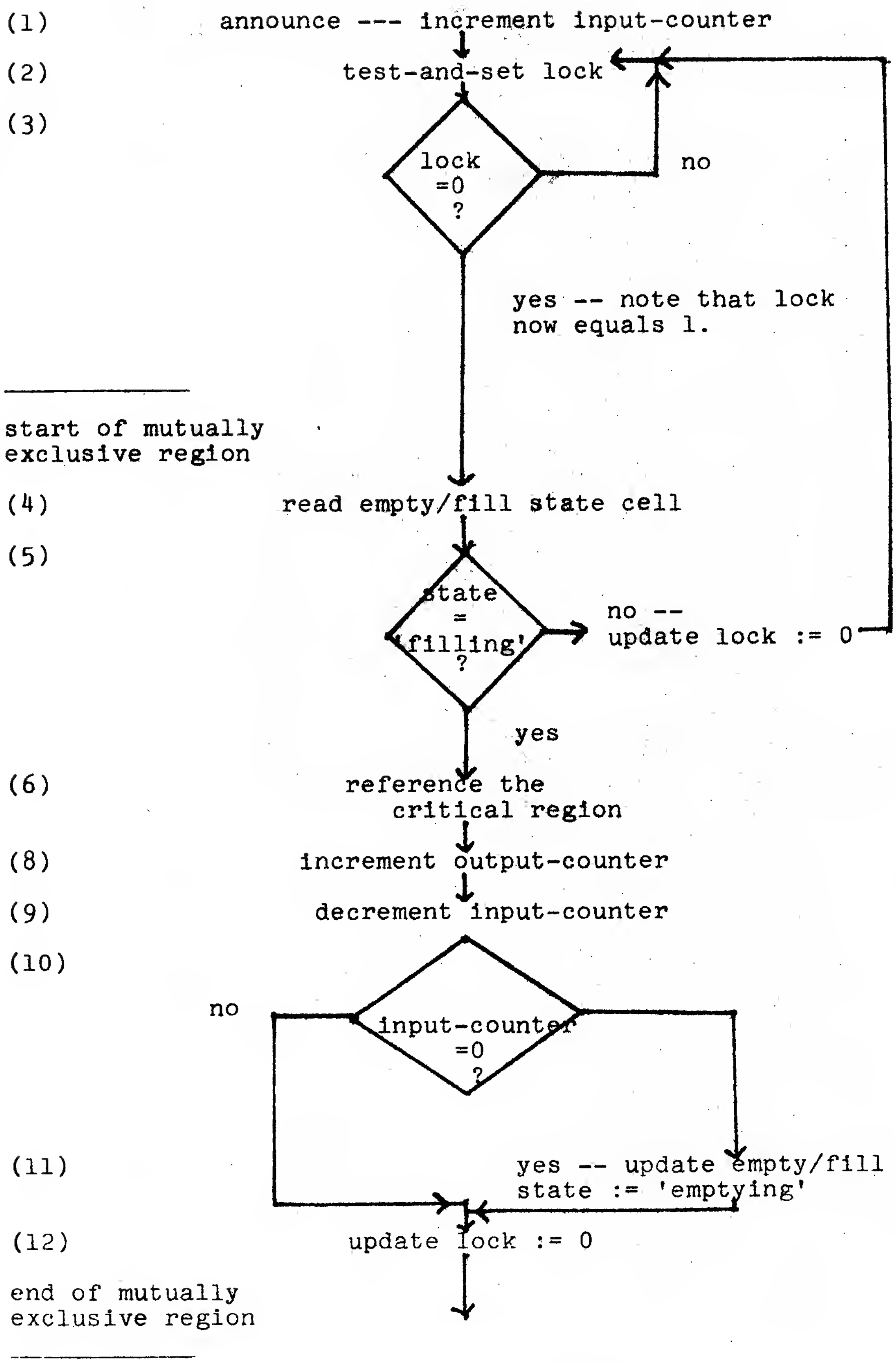
Consider first the problem of repeat sessions. We need to achieve the following specification: Suppose α_a and α_b are two processes waiting to pass through the mutual exclusion operator; and suppose that α_b makes it through before α_a. α_b must not attempt to pass through again until α_a passes through once. You may note that this specification is similar to the constraint satisfied by the scan algorithm presented in the previous section.

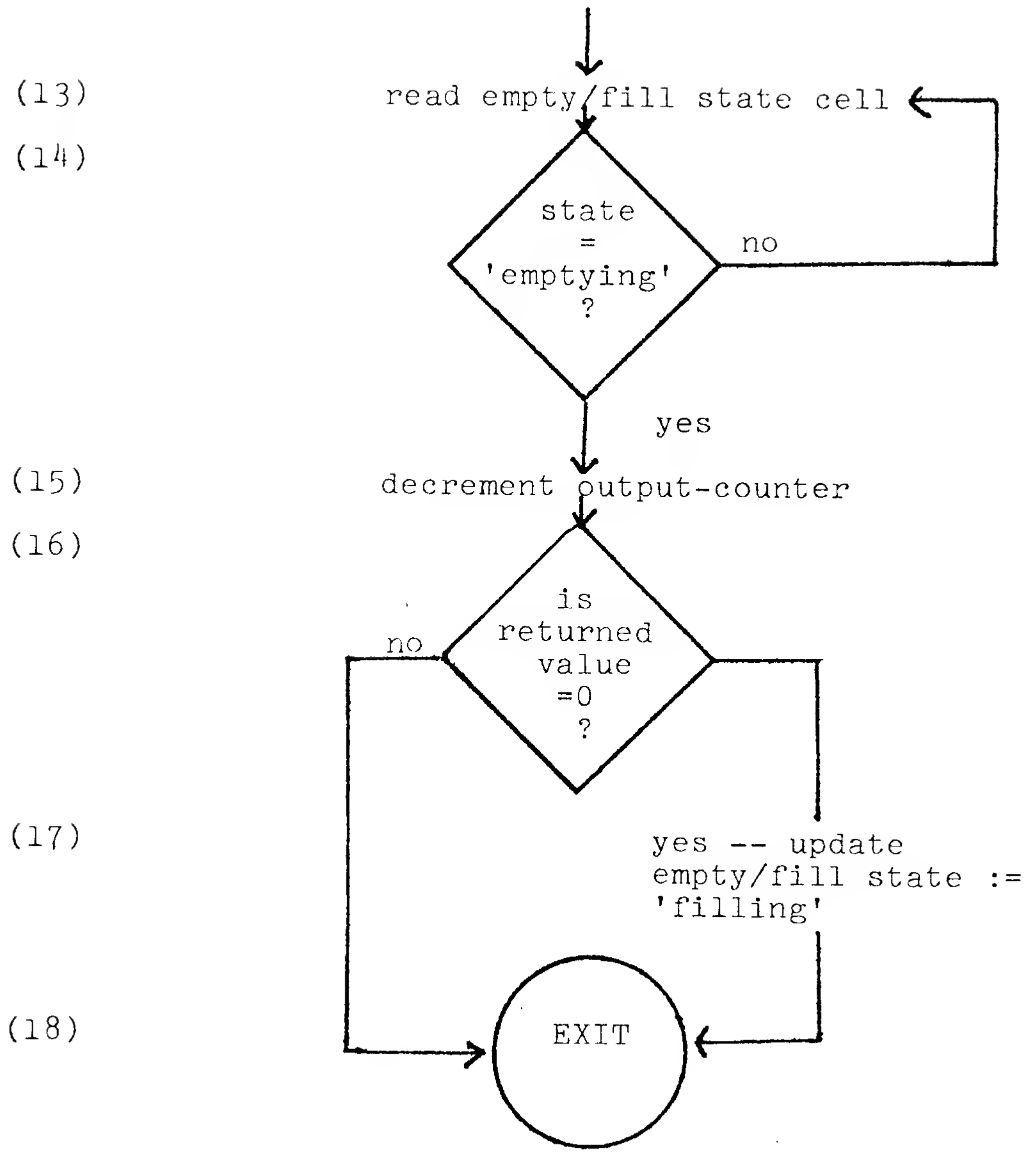
A sure way of keeping α_b from trying to reenter the competition is to keep α_b locked up inside the mutual exclusion device until α_a gets through. We imagine a device with two "chambers", an input chamber and an output chamber. Processes when they want to enter go into the input chamber to wait. They are allowed out of the input chamber one by one and they pass through the critical section. Afterwards, the processes are held up in the output chamber and made to wait some more until the input chamber is empty. See the illustration below:



A device like this has the potential for fairness if all the wait loops in both chambers can be shown to be of bounded duration.

An algorithm along these lines is flow-charted on the next two pages, and described thereafter: The key to the algorithm is that entering and exiting operations are decoupled; while processes are allowed to exit the device, none are allowed to enter. The cell, empty/fill-state, keeps track of whether the device is being filled or emptied.


(1)
announce --- increment input-counter
(2)
test-and-set lock
(3)
lock =0 ?
no
yes -- note that lock now equals 1.
start of mutually exclusive region
(4)
read empty/fill state cell
(5)
state = 'filling' ?
no -- update lock := 0
yes
(6)
reference the critical region
(8)
increment output-counter
(9)
decrement input-counter
(10)
input-counter =0 ?
no
(11)
yes -- update empty/fill state := 'emptying'
(12)
update lock := 0
end of mutually exclusive region


(13)
read empty/fill state cell
(14)
state = 'emptying' ?
no
yes
(15)
decrement output-counter
(16)
is returned value =0 ?
no
(17)
yes -- update empty/fill state := 'filling'
(18)
EXIT

fair-mutual-exclusion ≡ (assuming no new processes are created)

-- requires three read/modify/write type cells:

input-counter = number of processes either waiting to enter critical retion or in the critical region.

output-counter = number of processes waiting to exit operator

lock = 0 if no process is in critical region

= 1 if a process is in critical region

(1) announce desire to enter by incrementing the input-counter

(2) test-and-set lock, and let lock-state = returned previous value

(3) there are two cases for lock-state

(3-1) lock-state ≠ 0 -- repeat from step (2)

(3-2) lock-state = 0 -- proceed with step (4) -- note that the lock now equals 1 regardless

(4) read contents of a normal cell, called empty/fill-state, and let state = contents

(5) there are two cases for state:

(5-1) state = 'emptying' -- update lock = 0 -- i.e. free the lock and repeat from step (2)

(5-2) state = 'filling' -- proceed with step (6)

(6) process is now validly inside mutual exclusion operator; reference the critical region as required

(7) the process has finished referencing the critical region. It is now in the "output chamber"

(8) increment the output-counter

(9) decrement the input-counter, and let counter = returned value

(10) there are two cases for counter:

    (10-1) counter ≠ 0 -- skip to step (12) below

    (10-2) counter = 0 -- proceed with step (11)

(11) update empty/fill-state:= 'emptying'

(12) update lock:=0 -- i.e. free the lock

(13) read contents of empty/fill-state, and let state = contents

(14) there are two cases for state:

    (14-1) state = 'filling' -- repeat from step (13)

    (14-2) state = 'emptying' -- proceed with step (15)

(15) decrement the output-counter, and let counter = returned value

(16) there are two cases for counter:

    (16-1) counter ≠ 0 -- skip to state (18) below

    (16-2) counter = 0 -- proceed with step (17)

(17) update empty/fill-state:= 'filling'

(18) exit to the outside world.

Mutual exclusion per se is provided by the binary lock cell. The protection of the lock ranges from step (4) to step (12); only one process at a time will ever execute in that mutually exclusive region. You will note that the decoupling state variable, empty/fill-state is only referenced within that region and therefore there will never be any ambiguity as to the state of the device. We may study each of its modes separately.

In 'filling' mode, processes may arrive at both the input and output chambers of the device. In the input stage the processes will pile up in the loop of steps (2)-(3), accumulating there while other processes are executing in the critical section. At the output stage, processes also must wait, here in the loop of steps (13) and (14). This latter loop is controlled by the instructions at steps (9), (10) and (11), which control the empty/fill state of the device by examining the fullness of the input chamber.

During 'emptying', processes may also accumulate at the input gate to the mutual exclusion operator, however they may not enter the device. In this mode the processes that are looping through the output chamber peel themselves off and return to society at large. Thus liberated, any or all of the processes might immediately turn around and try to get back into the place, of course; however the decoupling of the input from the output due to the mode being 'emptying' prevents this feedback path from becoming oscillatory.

The fairness of this algorithm follows from the bounded duration of all its loops. There are three loops in the program:

(1) the loop at steps (2)-(3) where each process tries to grab the lock.

(2) the loop at steps (2)-(5) waiting for the state to be 'filling'

(3) the loop at steps (13)-(14) waiting for the state to be 'emptying'

We shall analyze these three loops and show each to be of bounded duration. We assume, of course, that the program starts out in its natural initial conditions: all counters = 0, the lock = 0, empty/fill = 'filling', and there are no processes inside the operator initially. Also recall that at this time we hypothesize that no new processes are created in the system; we will extend the solution to cover that case shortly.

Let's consider the loops in reverse order, starting with the loop waiting for the state to become 'emptying'. The end-test for the loop is satisfied when input-counter becomes 0. It is certainly the case that there is a maximum value attainable by input-counter: it may never exceed the number of processes existant in the system. Now, input-counter is decremented only within the mutually exclusive region which means that it may be decremented only while processes are allowed into that region. That is, input-counter may be decremented only while the mutual exclusion operator is in 'filling' mode. During this mode, though, no processes are allowed back out into the actor society where they may increment input-counter again. Thus during the phase when input-counter may be decremented, no process that has both incremented it and decremented it corresondingly may increment it again. Nor, of course, will any process increment the input-counter twice without decrementing it in between.

This implies that input-counter may be incremented only a bounded number of times per 'filling' mode. Once all processes have incremented it one, though its value may be far less than the maximum possible value, there will be no more increments until the mode switches to 'emptying'.

The activity of letting processes into the critical region will go on until there is a mode switch. Since there are only a bounded number of increments possible until the mode switches, and since each increment is uniquely matched with a decrement, the switching of the mode is inevitable and will happen in bounded time.

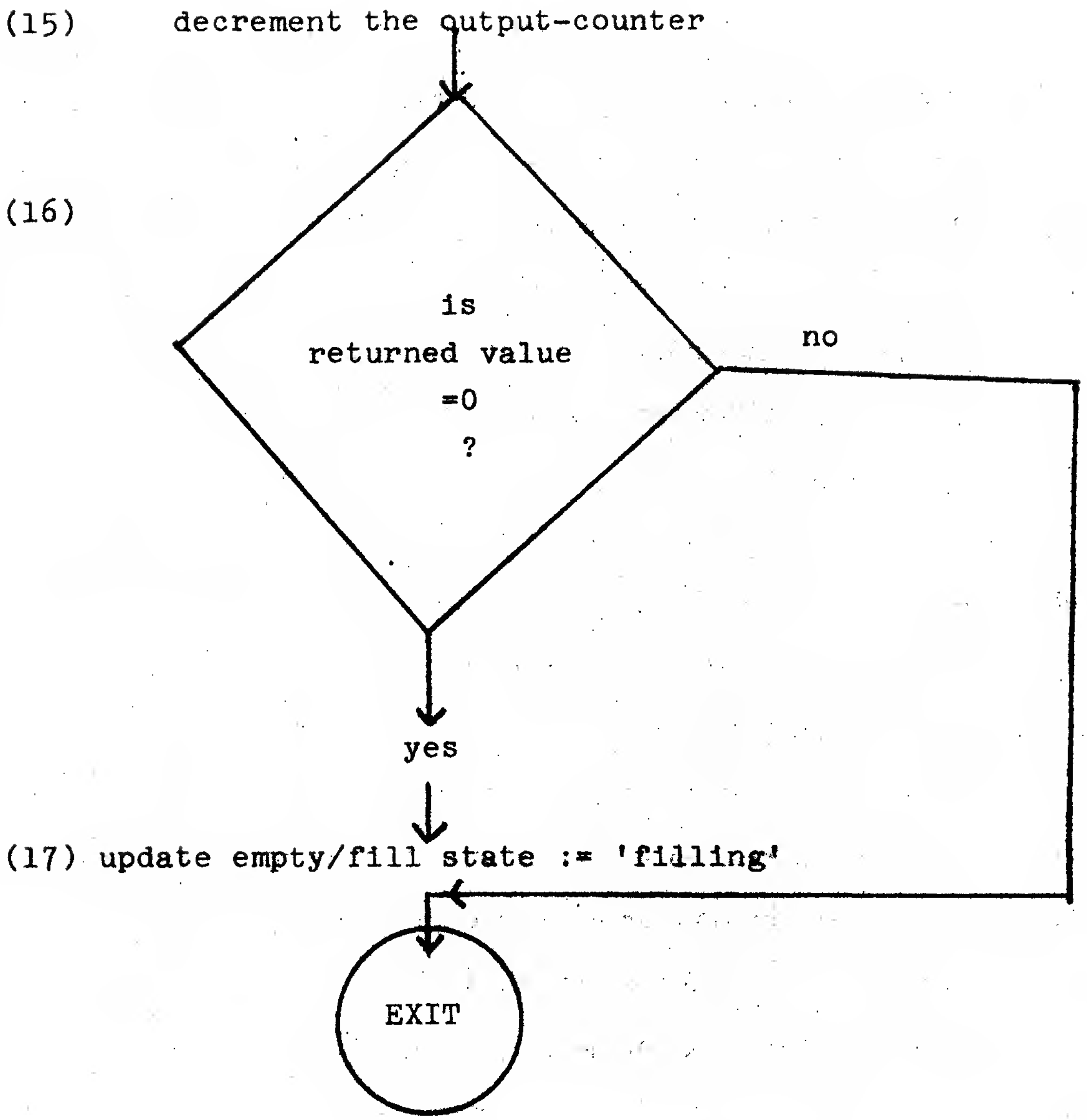
Thus loop (3) -- waiting for empty/fill-state to equal 'emptying' -- is a loop of bounded duration.

Now let's examine the next loop up the chain, the loop at steps (2) through (5) waiting for the state to switch back to 'filling'. The transition here is directly controlled by the activities of processes as they exit the output chamber in steps (13) -(18).

It is clear that while the output chamber is filling up, the value of output-counter will increase to a maximum value. The operation of emptying the chamber decrements the counter back to 0 while maintaining a decoupled relationship with the input of the device; during the emptying phase no processes are allowed to enter the output stage and so the output-counter is not subject to incrementing during that period. The operation of decrementing a number that is not being otherwise updated is an operation of bounded duration.

Therefore the second loop, also, is guaranteed to terminate in a bounded amount of time.

We should note before passing one important detail in the output algorithm, a fragment of which is reproduced below:


(15) decrement the output-counter
(16)
is
returned value
=0
?
no
yes
(17) update empty/fill state := 'filling'
EXIT

It is quite important that the value tested in step (16) is the value returned indivisibly by the decrement instruction, and that the value tested is not obtained by an independent read contents message. The algorithm as it is results in there being a unique event that observes output-count = 0 and hence a unique event that resets the mode to 'filling'. Were the test an independent activity, several different processes could decrement the counter before any tested it, and they all could observe output-count = 0.

It is not harmful in and of itself for several events to update state = 'filling' redundantly. If, however, some one of the processes tarried at length between step (16) -- the test -- and step (17) -- the update -- it might reset the state during the next pass of the program, after the state had just been set to 'emptying' for another go round. If so, the emptying of the output chamber would be aborted in mid-stream and if no more processes ever entered the device, the output chamber would never be emptied.

The algorithm as specified does not allow this potential deadlock.

Finally we reach the loop at the front door of the mutual exclusion device, the loop where processes vie with each other to grab the lock. This loop, of course, is the embryo of the whole machine, the sequence of events that assures mutual exclusion in the first place. The boundedness of this loop is implied by the bound on the loop encompassing steps (2) through (5); the larger loop encloses the former one.

We see then that the algorithm presented implements mutual exclusion and does so without introducing unbounded loops in the behavior of any process. The algorithm thus implements fair-mutual-exclusion as advertised. Furthermore it does so using a fixed number of cells, albeit cells extended to model read/modify/write instructions. This means that all synchronization primitives -- semaphores, serializiers, what have you -- may be implemented using just the normal, primitive memory arbitration schemes provided by most computer hardware, with no extra software-induced indivisibility of operation.

Other interesting equivalences of power may be demonstrated using this algorithm. The algorithm cannot be implemented using unfair semaphores and normal, unextended cells. This is because the announcement of arriving processes accomplished by incrementing input-counter in step (1) could not be assured with an unfair semaphore. But, the algorithm could work given unfair semaphores that will answer the question, "Are any processes at all waiting to get through you?" Given these rather trivially extended semaphores, all the counters in the algorithm would become obsolete, and the tests for zero would be replaced by the question. That is to say, this slightly extended unfair semaphore has equivalent power to the glorious fair semaphore.

The algorithm as presented will only work so long as the number of processes in the system remains bounded. If there is a process source out somewhere busily grinding out new processes, then the input-counter may be incremented forever and the mode switch from 'filling' to 'emptying' may never come about. In this case we would find stagnant pools of processes collecting in the output chambers of the locks in the system.

A way must be found to prevent newly created processes from competing with processes already in existence until the older processes get through the mutual exclusion device once. One way that this may be done is by restricting the actor system to have no more than one of the devices and insisting that all process creation happen behind that unique lock. Further, each newly created process must mimic its parent and wait in the output chamber until 'emptying' mode begins. This modification has the effect of keeping the set of processes that are candidates for entrance into the mutual exclusion device from acquiring any new members during any one filling session. The boundedness of the loops in the algorithm depends on this fact alone.

This solution restricting the system to one mutual exclusion operator is extremely inelegant and clearly inefficient. However it does work -- it does implement fair mutual exclusion for an arbitrary number of processes -- which is the major theoretical concern.

4. Conclusion

We have presented two algorithms that implement fair mutual exclusion for an arbitrary number of processes. Both algorithms use relatively simple synchronization primitives; the first solution uses cells and the second uses cells extended so as to model read/modify/write instructions.

The cell based solution utilized an array of cells with one cell per process. Similar algorithms have appeared in the literature previously as we have noted; the unique contribution that we make is to show how the algorithm may be generalized from an array to a variable size date structure. We presented an algorithm for expanding that structure through the addition of newly created processes and proved that fair mutual exclusion could be retained by the algorithm evenif the number of processes in the system were to grow without bound over time.

We proved that the fairness of the solution in the face of proliferating processes depends critically on where the new processes are appended to the structure of cells. In our solution there is a definite order to the cells in the structure and requests from processes to pass through the operator are serviced via a scan algorithm that scans from the first to the last cell in that order. If the new processes are added at the end of the data structure -- i.e. after the last cell -- then the scan algorithm could get "stuck" in the expanding, new portion of the structure. This would happen if new processes were being added and each put in a request to pass through at a faster rate than requests were being serviced.

However, if the new processes are inserted at the beginning of the data structure -- i.e. before the first process's cell -- then the scanner cannot get stuck. Whenever the scan algorithm finishes the last process it must of course reset itself to the first process, thus wrapping around. We said that a new cycle of the scan began everytime the scan were reset in this manner. The important fact is that if new processes are added at the beginning of the structure, the size of each cycle is some particular fixed number. Two different cycles may very well be of different sizes, but once a cycle begins its size does not change. Therefore once a cycle begins we may be sure that it will end in a bounded number of steps and a new cycle begun thereafter.

The fairness of the scanning operation follows from the boundedness of each cycle. Because that means that everytime some particular process is scanned it will be scanned again in the future within a bounded number of events.

The other point to be careful of in expanding the data structure is to make sure that only one process expands it at a time. That is, the operation of expanding the cell structure to accomodate newly created processes must itself be done in a mutually exclusive fashion.

The second fair mutual exclusion solution that we presented used read/modify/write type cells as the synchronization primitive. It is well known how to implement unfair mutual exclusion with these extended cells, using a test-and-set instruction and a binary lock variable. We prove that it is possible to achieve fair mutual exclusion also using a small number of these extended cells; in particular the number of cells required is much less than the number of processes in the system.

However a serious deficiency of this algorithm is that it introduces considerable delay in the execution of programs beyond that required for mutual exclusion per se. Mutual exclusion algorithms may always delay processes that are trying to enter the critical, protected region; our second algorithm here, though, also delays the processes as they try to exit from the device.

From a theoretical standpoint it is interesting that fair mutual exclusion of an arbitrary number of processes may be implemented using such simple primitives. In this sense more sophisticated primitives such as serializers have no more power than simple little cells. But from a practical point of view differences do emerge. Both algorithms that we present have efficiency related drawbacks: The cell solution requires lots of memory -- it needs one cell per process per mutual exclusion operator. The extended cell solution is slow -- it introduces approximately twice as much delay on average than is required by mutual exclusion per se. So while primitives like cells are complete synchronization primitives and a theory does not need more elaborate primitives in order to coordinate parallel processes, cell solutions are inefficient. More sophisticated synchronization primitives are desirable for this reason.

BIBLIOGRAPHY

Bobrow, Daniel G. and B. Wegbreit (1973), A Model and Stack Implementation of Multiple Environments, Comm. of the ACM, Vol. 16, No. 10, pp 591-603, October, 1973.

de Bruijn, N.G. (1968). Additional Comments on a Problem in Concurrent Programming Control, Comm. of the ACM, Vol. 10, No. 3, pp 137-138, March, 1968.

Burstall, Rod M., J.S. Collins, and R.J. Popplestone (1971). Programming in POP-2, Edinburgh University Press, 1971.

Courtois, P.J., F. Heymans, and D.L. Parnas (1971). Concurrent Control with "Readers" and "Writers", Comm. of the ACM, Vol. 14, No. 10, pp 667-668, October 1971.

Dijkstra, Edsger W. (1965). Solution of a Problem in Concurrent Programming Control, Comm. of the ACM, Vol. 8, No. 9, p 569, September 1965.

Fennel, R.D. and V.R. Lesser (1975). Parallelism in A.I. Problem Solving: A Case Study of Hearsay II, Department of Computer Science, CMU, October 1975.

Goodman, Nat and C.C. Goodman (1973). Introduction to LOGO Geometry, Bionics Research Report, School of Artificial Intelligence, University of Edinburgh, June 1973.

Greif, Irene (1975). Semantics of Communicating Parallel Processes, MAC-TR-154, M.I.T., September 1975.

Greif, Irene and C. Hewitt (1974). Actor Semantics of PLANNER-73, Artificial Intelligence Laboratory Working Paper # 81, M.I.T. November 1974.

Habermann, A. Nico (1972). Synchronization of Communicating Processes, Comm. of the ACM, Vol. 15, No. 3, pp 171-176, March 1972.

Hewitt, Carl (1974). Protection and Synchronization in Actor Systems, Artificial Intelligence Laboratory Working Paper # 83, M.I.T., November 1974.

Hewitt, Carl (1975). How to Use What you Know, Artificial Intelligence Laboratory Working Paper # 93, May 1975.

Hewitt, Carl and R. Atkinson (1976). Synchronization in Actor Systems, draft, April, 1976.

Hill, J. Carver (1973). Synchronizing Processes with Memory-Content-Generated Interrupts. Comm. of the ACM, Vol. 16, No. 6, pp 350-351, June 1973.

Knuth, Donald E. (1966). Additional Comments on a Problem in Concurrent Programming Control, Comm. of the ACM, Vol. 9, No. 5, pp 321-322, May 1966.

Misunas, David P. (1975). A Computer Architecture for Data-Flow Computation, S.M. Thesis, Department of Electrical Engineering and Computer Science, M.I.T. June 1975.

Parnas, David L. (1969). On Simulating Networks of Parallel Processes in Which Simultaneous Events May Occur, Comm. of the ACM, Vol. 12, No. 9, pp 519-531, September 1969

Pratt, Terrence W. (1975). Programming languages: Design and Implementation, Prentice-Hall Inc., Englewood Cliffs, N.J., 1975.

Presser, Leon (1975). Multi-programming Coordination, Computing Surveys, Vol. 7, No. 1, pp 21-44, March 1975.

Schlageter, Gunter (1975). Access Synchronization and Deadlock Analysis in Database Systems: An Implementation Oriented Approach, Information Systems, Vol. 1, pp 97-102, Pergamon Press, 1975.

Shoshani, A. and A.J. Bernstein (1969). Synchronization in a Parallel-Accessed Database, Comm. of the ACM, Vol. 12, No. 11, November 1969.

Stansfield, James L. (1975). Programming a Dialogue Teaching Situation, Ph.D. Thesis, University of Edinburgh, January 1975.

Weng, Kung-Song (1975). Stream Oriented Computation in Recursive Data Flow Schemas, S.M. thesis, Department of Electrical Engineering and Computer Science, M.I.T., September 1975.

CS-TR Scanning Project
Document Control Form

Date : 11 / 3 / 95

Report # LCS-TR-173

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:

Document Information

Number of pages: 120 (126-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☒ Unknown ☐ Other:

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other:

Page Data:

Blank Pages(by page number):

Photographs/Tonal Material (by page number):

Other (note description/page number):

Description : Page Number:

IMAGE MAP! (1-5) UN#'ED COVER, BLANK, ABSTRACT, BLANK, TABLE CONTENTS,
(6-120) PAGES #'ED 2-116
(121-126) SCANCONTROL, COVER, SPINE, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 11 / 3 / 95 Date Scanned: 11 / 24 / 95 Date Returned: 11 / 27 / 95

Scanning Agent Signature: Michael W. Cook